LECTURE NOTES. CSI 32. INTRODUCTION TO COMPUTER PROGRAMMING 2

BASED ON *OBJECT-ORIENTED PROGRAMMING IN PYTHON, BY GOLDWASSER AND LETSCHER*

(http://cs.slu.edu/∼goldwamh/oopp/)

1. Data and Types: Functions and Algorithms, Objects and Classes: OO Design (sections 1.1, 1.2, 1.3, 1.4, 1.5)

- Two aspects of computing: Data and Operations.
- **Data and Types:**
- Definition of Data in the context of computing:
- The quantities, characters, or symbols on which operations are performed by computers and other automatic equipment, and which may be stored or transmitted in the form of electrical signals, records on magnetic tape or punched cards, etc.,.
- Within modern computers, all information is represented as a collection of binary digits, also known as **bits.**
- Each bit can be set to 0 or 1.
- Combining many bits a wider variety of patterns can be stored to represent pieces of information.
- A **byte** is a set of 8 bits. It can represent $2^8 = 256$ different values. With millions of bits, large amount of data can be stored.
- Much of the data being used by a computer processor is stored in the computer's **main memory**, referred to as RAM (random access memory).
- **Information** is high-level abstraction, while **data** is a low-level representation of such information, capable of being stored and manipulated by a computer. That is, there could be several ways of representing information – each such choice of representation is an **encoding** (example, JPG and GIF formats are different encodings of the same picture, or the same information).
- **Data types:**
- Though all data is stored as a sequence of bits, it would be cumbersome to specify the exact encoding for every piece of data.
- Higher-level abstractions are supported through the definition of **data types.**
- Example, integer, floating point, string, or user-defined types.
- Numbers, characters, dates, lists exist as pre-defined data types. These are often called the **primitive** or **built-in** data types.
- **User-defined types** are built by programmers who may want data abstractions that are specific to his/her own implementation. These are **Classes** in Python.
- **Operations, Functions, and Algorithms:**
- The hardware component that **controls** the entire computer is known as its **central processing unit (CPU)**. Commonly supported instructions are: loading data from main memory into the CPU, storing data from CPU to the main memory, performing basic arithmetic operations or checking basic boolean conditions.
- Just as data types can be built-in or user-defined, operations are also built-in or user-defined.
- In addition, there are **control structures** which determine when other instructions are executed or repeated (eg. if-elif-else, for-loops, while-loops).
- **Algorithms:**
- A step-by-step procedure for solving a problem or accomplishing some end especially by a computer is called, an algorithm.

- Example: Find the greatest common divisor (gcd) of two positive natural numbers.
- Here is a flow-chart – discuss this algorithm.

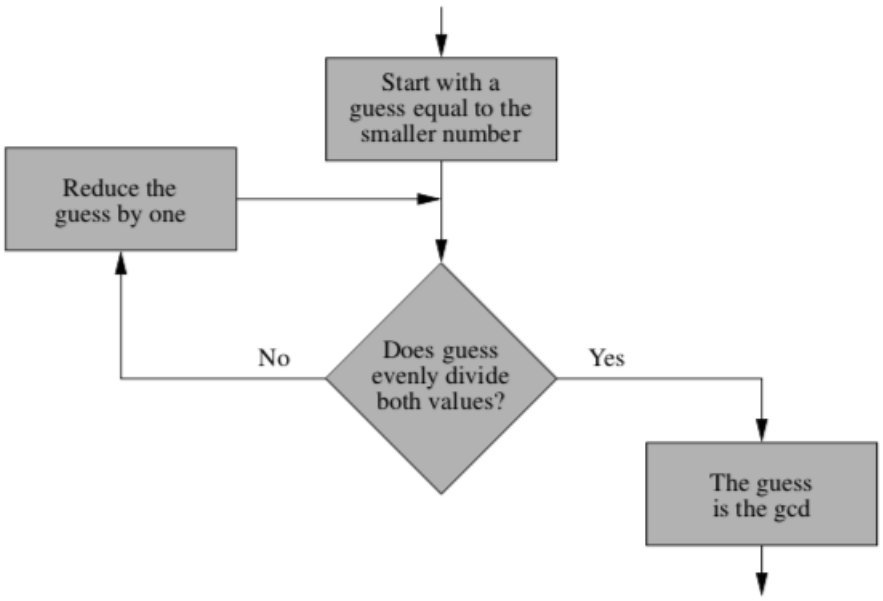


FIGURE 1.1: Flowchart for an algorithm that computes the gcd of two integers.

- Here a flow-chart of the **Euclid's algorithm** for computing the gcd of two non-negative integers, both non-zero. Go through this procedure for at least two concrete examples.

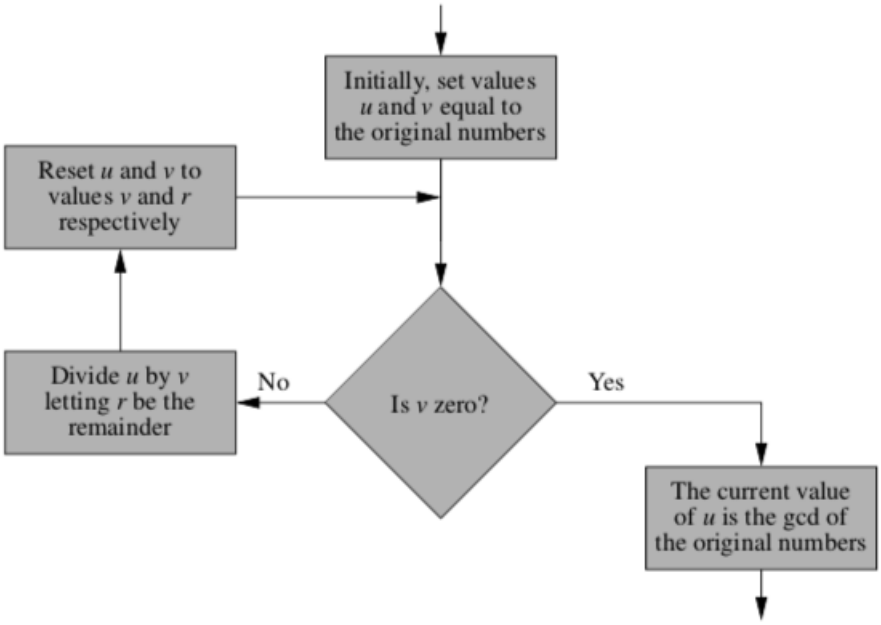


FIGURE 1.2: Euclid's algorithm for computing the gcd of integers u and v.

- **High-Level Programming Languages:**
- Every piece of software is executed by the computer's CPU.
- The CPU provides core support for the most primitive data types (eg. integers, floats) and instructions (eg. addition).
- Each CPY has its own programming language called, **the machine language.**
- Because a machine language supports only the most basic data types and operations, it is often described as a **low-level programming language.**
- Software is typically developed with **high-level programming languages.**
- The computer's hardware only understands the low-level language. A programmer programs in a high-level language and creates one or mote text files that are called **source code**.
- In order to be executed, that high-level source code gets translated into equivalent low-level instructions.
- These translators are in two forms: **compilers** and **interpreters.**
- A compiler takes the entire original code and translates it into a low-level program known as an **executable.** This executable can then be run directly by the computer.
- An interpreter proceeds piecewise, translating an instruction, executing it, then translating the next instruction, and so on.
- There are hundreds of high-level programming languages. Some of the most commonly used ones: Ada, C, C++, C#,Cobol, Fortran, Java, etc.,.
- Each language has its own **syntax** and **semantics**.
- The syntax of a language is the precise set of rules regarding how character, words, and punctuation should be used.
- A compiler or interpreter reports a **syntax error** if your code is not formatted according to the rules of the language you use.
- The **semantics** mean the underlying meaning of statements written in your code.
- If a code is written which is syntactically correct, but does not produce the desired result, then most likely there is a semantic error. For instance, if you want the quotient when 12 is divided by 5, you should type in 12//5 rather than 12/5 (what is the difference?).
- **Types of Errors:**
- **Syntax Error:** Try the following on your Python shell:

```
print("Hello)
```

- Compare it with

```
print("Hello.")
print("How are you?)
```

- **Runtime Error:** Here, the syntax is correct but the interpreter is unable to run your code to completion. For example:

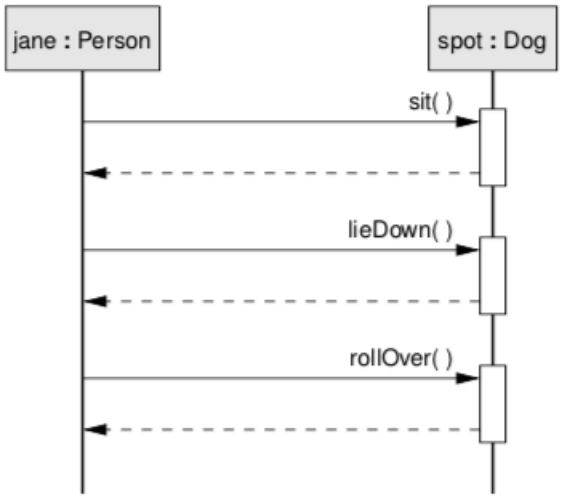
```
print(1/0)
```

- **Logical Error:** Here, the syntax is correct, and the interpreter is able to run your code to completion, but you made a logical error at some point. For example,

```
print("2+2 +10")
```

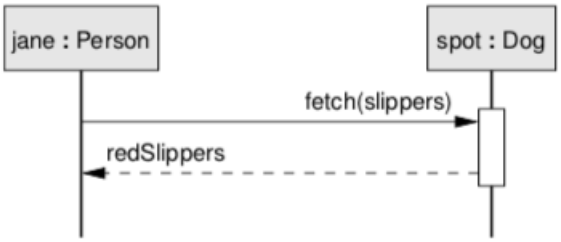
- Logical errors are the hardest to find because it is the author of the code who is responsible for the error. Python will find its own errors, but the author of the code will have to find his/her own errors.
- **Please do the following now:**
- Sign up on *codelab*.
  REGISTRATION (FOR STUDENTS):
  1) Go to www.turingscraft.com
  2) Click "Register for CodeLab" and follow the instructions.
  3) When prompted, enter the Section Access Code: UNAB-26799-HSGP-39
- If you wish to use your home computers, download Python 3.7 from www.python.org. You will see versions for various Operating Systems. Choose the appropriate installer. Run the installer and click through the prompts (default options are fine). The application "IDLE" will be installed by default.
- After installation you can run IDLE by clicking:
  Start → All Programs → Python 3.7 → IDLE (Python GUI)
  Or, if you are on Mac, go to
  Applications → Python 3.7 → IDLE
- Try IDLE. Our lab computers have IDLE set up.
- An easy but limited option is Brython. Check out, https://brython.info/tests/editor.html
- Another option is to use *jupyter*: Check out, https://jupyter.org/install
- Sign up for Dropbox at https://www.dropbox.com.
  (Do not use https://www.dropbox.com/business – this one requires payment).
- We will be using Dropbox for homework and test submissions. Create a folder "Spring2019-CSI32-Assignments-FirstName-LastName" and share it with uma.iyer@bcc.cuny.edu

- **The Object-Oriented Paradigm:**
- Object Oriented Programming (OOP) is the process of modeling data and operations in a paired form, rather than as separate entities.
- Objects of the same class share a common encoding of their underlying data representation and support a common set of operations.
- A single object from a given class is termed, an **instance** of that class.
- Internally, each instance is represented by one or more pieces of data, called **attributes**, or **data members**, or **fields**, or **instance variables**.
- For instance, *jane = Person('F', 25, 32000)*, is an assignment where *jane* is assigned an instance of the *Person* class. This *object* possibly has instance variables gender (jane.gender = 'F'), age (jane.age = 25), anninc (the annual income, jane.anninc = 32000).
- Note, *jen = Person('F', 25, 32000)*, is another assignment, where *jen* is assigned an instance of the *Person* class. Note, a **new object** is created when we invoke the constructor *Person* again even when the attributes may be identical.
- Taken together, the attribute values of a particular instance comprise the **state information**. Note that, over time, the state information may change.
- The operations suported by instances of a class are known as **methods**, or **actions**, **behaviors**, or **member functions**.
- For instance, *jane.vote()*, *jen.info()*, *jane.addincome(2000)*, or *jen.addage(3)* are various methods with or without parameters.
- Collectively, the attributes and methods of an instance are called its **members**.
- Note that objects can interact with other objects. Say, in addition to the *Person* class, we also had a *Dog* class.
- Here is a **sequence diagram** demonstrating interactions between jane and spot (an instance of the Dog class). Note that *jane* is the **caller** and *spot* is the **callee**.



- The vertical line drawn down from the label represents the chronological lifespan of that object.
- Each solid arrow represents the invocation of a method, oriented from the caller to the callee.
- When a method is called, we say that the flow of control is passed from the caller to the callee.At that time, the caller waits until the callee completes the action.

- A white rectangle on the callee's side denotes the duration of time when tat action is being performed. At its conclusion, the flow of control is passed back to the caller, as diagrammed with a dashed line in our figure.
- Here is another sequence diagram:

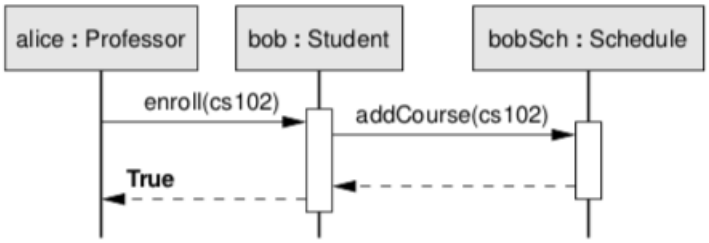


- This represents calling a method with a parameter. Presumably, spot.fetch(slippers) returns a value.
- The overall agreement regarding the expected parameters and return value for a method is called its **signature**.
- Some methods do not cause any change in the object's current state, but are used by the caller to request information about that state. Such methods are commonly termed **accessors** or **inspectors** and necessitate the use of a return value. Example, jane.getage()
- A method that affects the state of the callee is termed a **mutator**. For example, jane.addage(3).
- Some methods are neither accessors nor mutators. For instance, spot.fetch(slippers) presumably does not change the values of instance variables of spot.
- **The Design of a Television Class:**
- An object from the Television class is an instance of this class.
- When an object is created, the system sets aside a particular chunk of memory for storing the attributes values that represent the state of the new instance. Some attributes do not change (example, brand and model of a television), while some other attributes are changeable (example, volume and channel).
- The desired behaviors of a television object must include volumeUp, volumeDown, mute, channelUp, channelDown, setChannel to a number etc.,.
- Here is a Television class diagram:

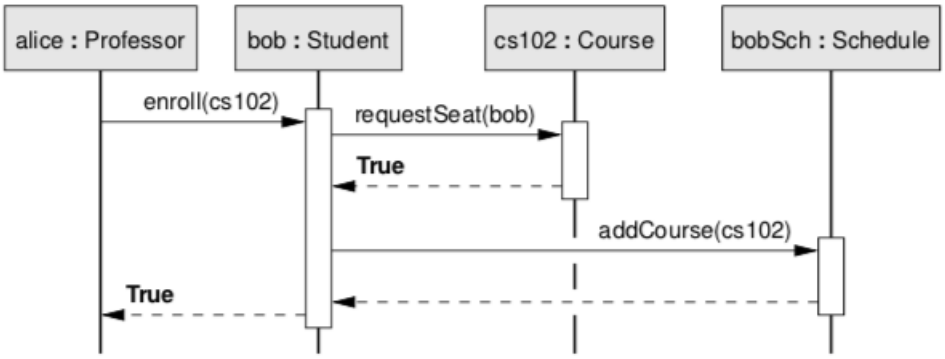
Television	
brand model powerOn	volume muted channel prevChan
togglePower( ) volumeUp( ) volumeDown( ) toggleMute( )	channelUp( ) channelDown( ) setChannel(number) jumpPrevChan( )

- Note that toggleMute does not simply set volume to zero. The mute button when pressed twice, should return volume to what it was before the TV got muted. Thus, the attribute muted is set either to True or False, while toggleMute method reverse the setting for the muted attribute.

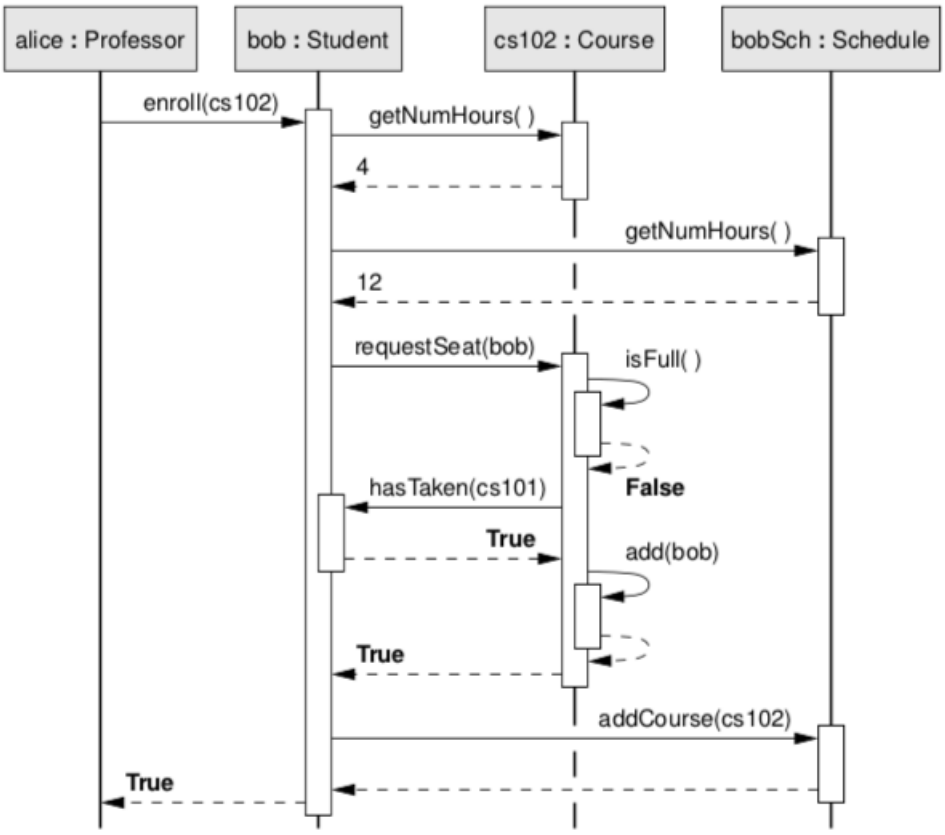
- We will return to this example later in the course.
- **The Design of a Student Registration System:**
- Note that the Television class was self contained. Typically though, we have interactions between objects from various classes.
- For example, here are classes where various objects are expected to interact: Student, Department, Course, Professor, Schedule, Transcript, and Major.
- There are relationships between these objects. For instance, a Course has. teacher, a Schedule has courses.
- Here is a sequence diagram for invocation of bob.enroll(cs102). Note, bob.enroll(cs102) in turn triggers bobSch.addCourse(cs102). The former is completed only after the latter is reutrned.



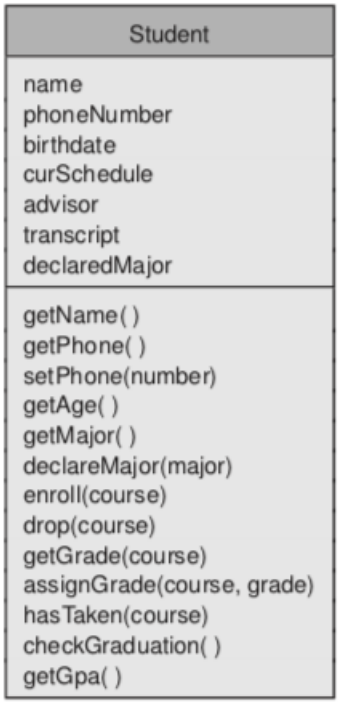
- In reality, the picture might be more complicated. For instance, here is a refined sequence diagram for invocation of bob.enroll(cs102):

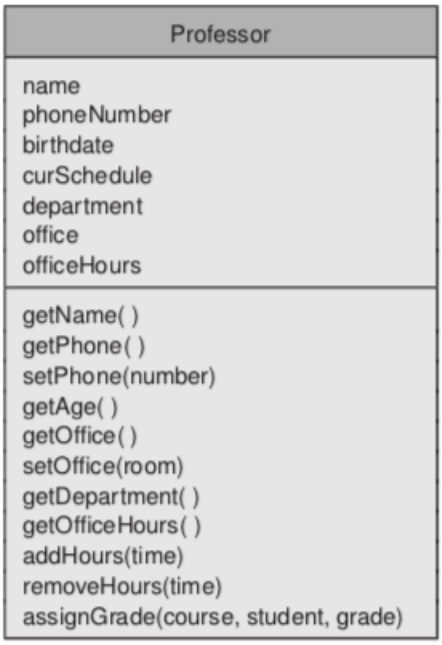


- Here is a sequence diagram with more steps (more realistic):

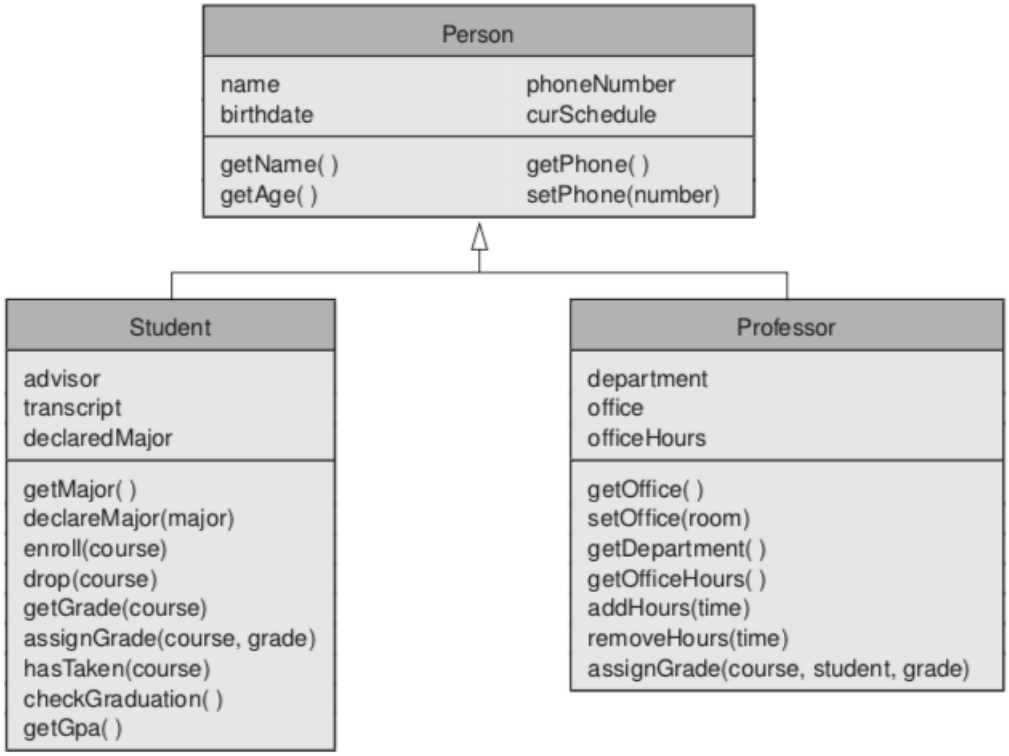


- **Class Hierarchies and Inheritance:**
- As we start the modeling of different classes, we should consider how those classes compare to one another.
- By identifying commonalities, we can organize a **class hierarchy.**
- Here are class diagrams for Student and Professor classes.





- Now here is a **hierarchy** showing Student and Professor classes derived from Person class:



- Note that the Person class is a more generic class than Student and Professor classes, based on their commonalities.
- When modeling the Student class relative to thePerson class, a Student instance **inherits** all of the attributes and methods of the Person class. Similarly, the Professor class inherits from Person and then additional attributes and methods are specified.
- The Person class is the **parent class** or the **base class** of Student class (and Professor class).
- The Student class (or the Professor class) is the **child class** or **subclass** of Person class.
- The relationship between a parent and child class is often termed an **is-a relationship**, in that every Student is a Person, but the converse is not true. Every Professor is a Person, but the converse is not true.
- Notice the difference between an **is-a relationship** and **has-a relationship**. A Student has a Transcript, a Course has a Professor (no inheritance is implied). In a **has-a relationship** a class has attributes that are instances of another class.
- **A drawing package:**
- We consider the design of a system for creating and displaying a variety of graphical objects.
- Start our design with a generic **Drawable** class.
- While we expect to be able to create c = Circle(Point(5,5), 3), we do not expect to create d = Drawable(...).
- Such a class that is designed to serve as a parent class in our inheritance hierarchy, unifying the common traits and behaviors, is known as an **abstract class**.

- The design of our Drawable class is given here (what do you expect the attributes and methods to mean?):

Drawable		
depth	transformation	referencePoint
move(dx, dy) moveTo(x, y) getDepth( ) setDepth(depth)	rotate(angle) scale(factor) flip(angle) clone( )	getReferencePoint( ) adjustReference(dx, dy) *draw( )*

- Note that within drawable objects, some are fillable and some not. Some drawable objects have no shape (example texts) and some have shapes. There are many such considerations one has to think about before we create our hierarchy. It helps to write down the design structure before classes are built.

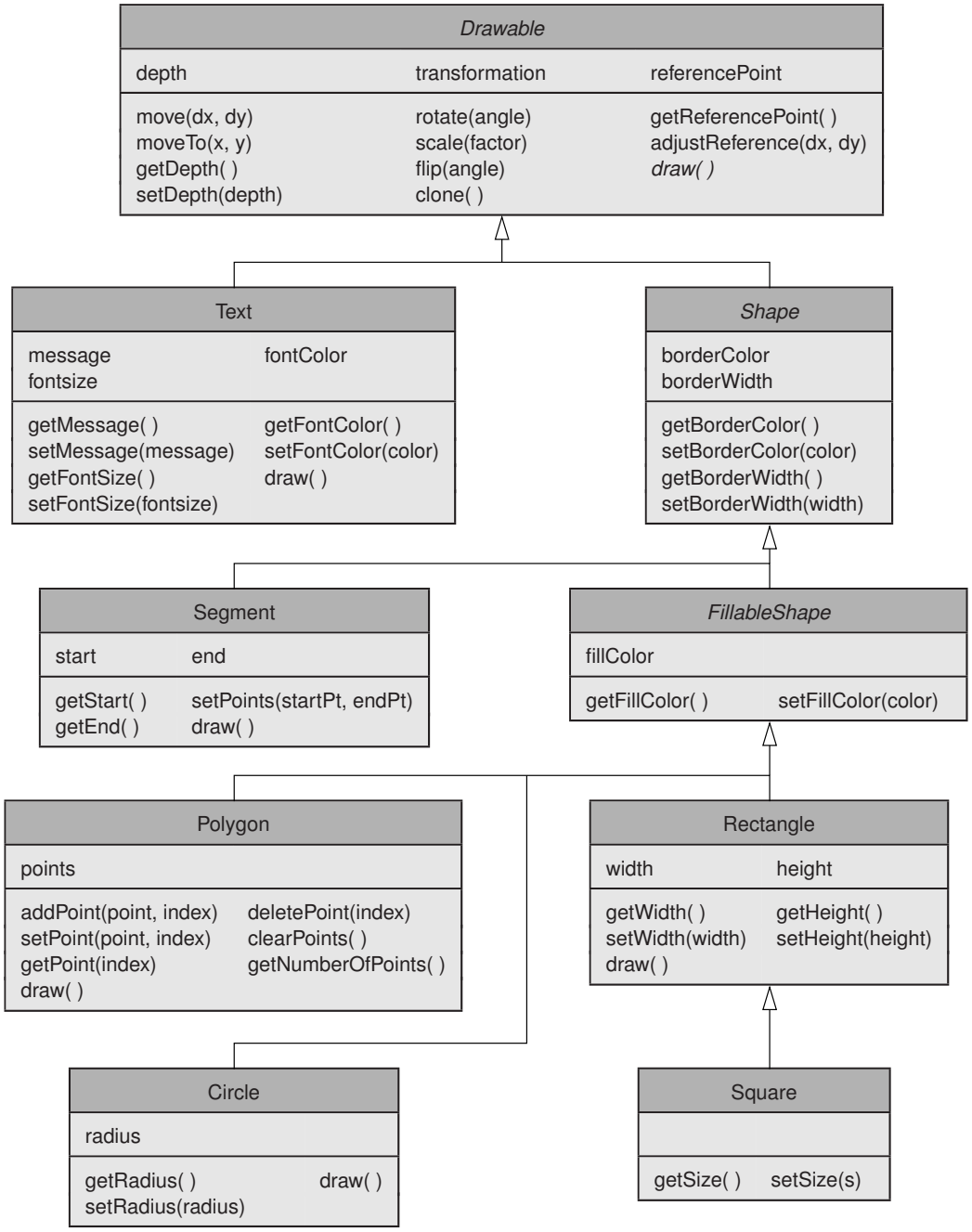


FIGURE 1.13: A proposed hierarchy of drawable objects.

2. Unified Modeling Language (UML)

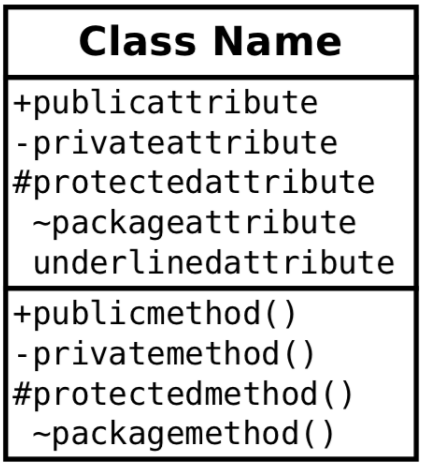
- The **Unified Modeling Language (UML)** is a standard visual modeling language intended to be used for *modeling business and similar processes,* and *analysis, design, and implementation* of software-based systems.
- UML is a common language for business analysts, software architects and developers used to describe, specify, design, and document existing or new business processes, structure and behavior of artifacts of software systems.
- UML is a way of visualizing a software program using a collection of diagrams.
- The notation has evolved from the work of Grady Booch, James Rumbaugh, Ivar Jacobson, and the Rational Software Corporation to be used for object-oriented design.
- It has since been extended to cover a wider variety of software engineering projects.
- Today, UML is accepted by the Object Management Group (OMG) as the standard for modeling software development.
- The current UML standards call for 13 different types of diagrams, classified into two distinct groups:
- **Structural UML Diagrams (or the Static View)**: Structure diagrams emphasize the things that must be present in the system being modeled. Here we have the following:
  (1) Class diagram
  (2) Package diagram
  (3) Object diagram
  (4) Component diagram
  (5) Composite structure diagram
  (6) Deployment diagram
- **Behavioral UML Diagrams (or the Dynamic View)**: Behavior diagrams emphasize what must happen in the system being modeled. Here we have the following:
  (1) Activity diagram
  (2) Sequence diagram
  (3) Use case diagram
  (4) State diagram
  (5) Communication diagram
  (6) Interaction overview diagram
  (7) Timing diagram
- To learn more about UML refer to
  https://www.smartdraw.com/uml-diagram/
  https://www.uml-diagrams.org/
  https://en.wikipedia.org/wiki/Unified_Modeling_Language
- In our course we will learn to draw the most basic forms of **Class diagram, Activity diagram, Sequence diagram**, and **State diagram**. We will use *Dia Diagram Editor* to draw these diagrams. Please download Dia from http://dia-installer.de/download/index.html for your home computer.

- **Class Diagram**
- A class diagram is a static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (methods), and the relationships among objects.
- A class is represented by a rectangle containing three compartments:

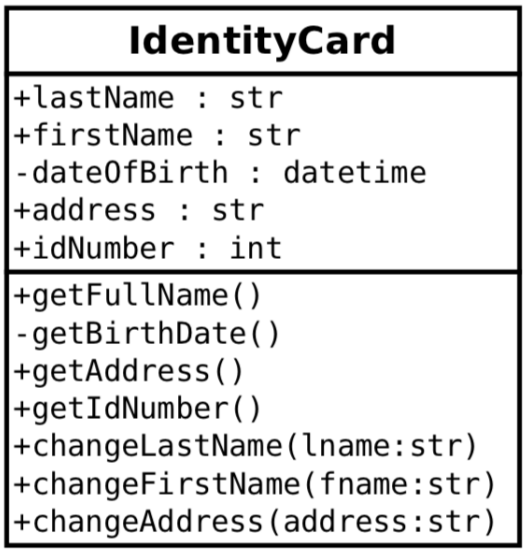
  The top compartment contains the name of the class. It is printed in bold and centered, and the first letter is capitalized.

  The middle compartment contains the attributes of the class. They are left-aligned and the first letter is lowercase. The attributes have specific visibility (Public, Private, Protected, Package).

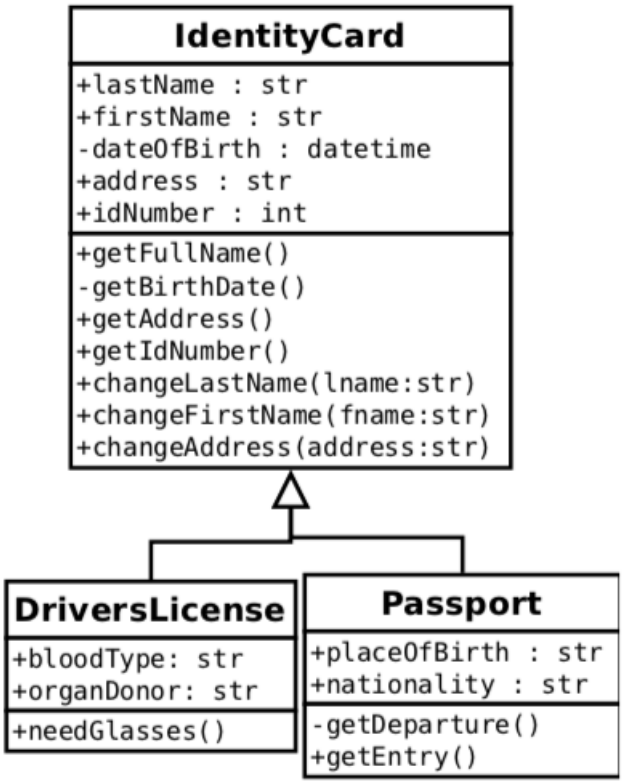
  The bottom compartment contains the operations the class can execute. They are also left-aligned and the first letter is lowercase. Methods are also assigned visibility.
- Here is a template of a class:



- **Visibility:** *Public* visibility means the attribute can be accessed from everywhere. *Private* attribiutes can be accessed only from within the class. *Protected* attributes can be accessed by the class and its children classes, while *Package or Default* attributes can be accessed by any class within the package of your class.
- For a concrete example:

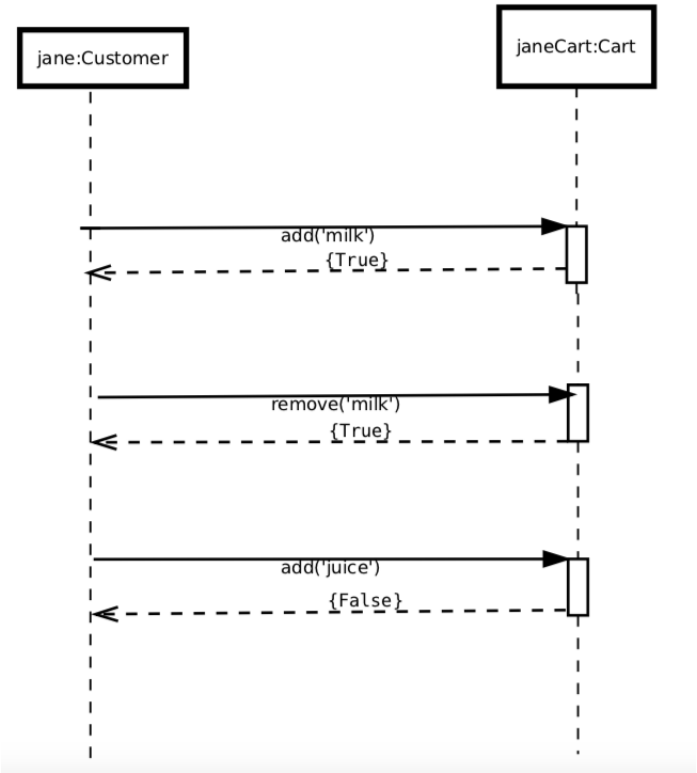


- **Inheritance** is denoted by a hollow arrow starting from a child class to its parent class.
- Here is a concrete example:



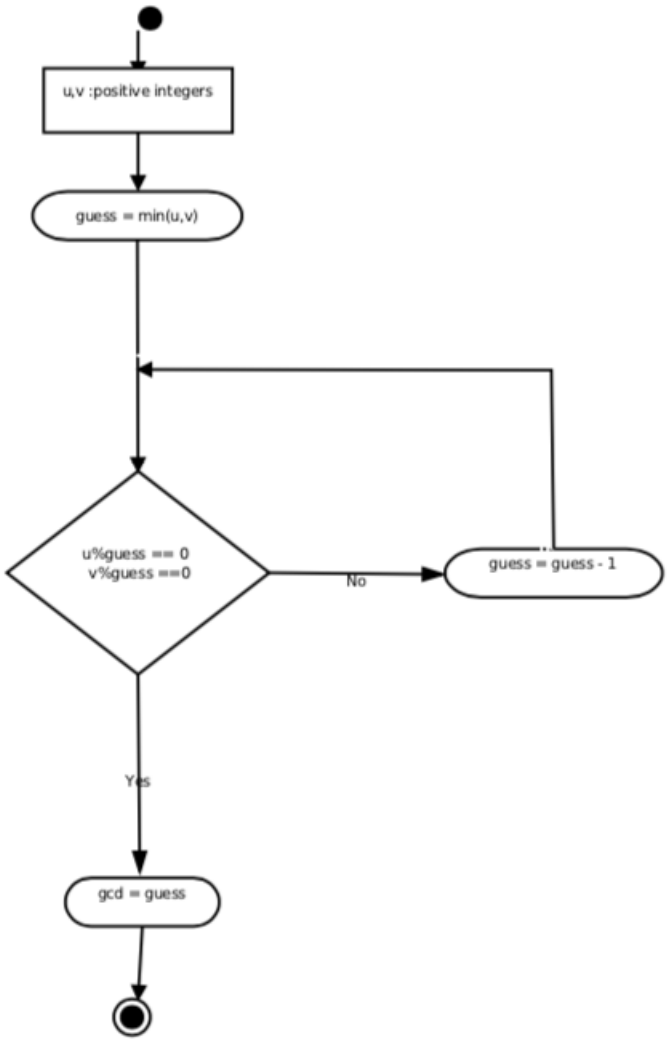
- As an example, pretend that you are a business owner with some employees and clients.
- Design an abstract class *Person*, with some basic attributes and methods.
- Now design two child classes, **Employee**, and **Client** with their respective attributes and methods.
- Draw a UML diagram to represent these interactions.

- **Sequence Diagram**
- A sequence diagram (or an event diagram or event scenario) is a visual depiction of interactions (methods) between objects.
- Vertical lines (lifelines) are drawn under each object involved in the interaction. Horizontal arrows represent the messages exchanged between them, in the order in which they occur.
- Here is a simple example:



- Now, draw a sequence diagram depicting the interaction between three objects *jane:Employee*, *mika:Client*, and *mikaOrder:Order*.
  (1) Jane asks Mika for Mika's order.
  (2) Mika adds to her order a salad.
  (3) Mika's order returns True.
  (4) Mika adds to her order a wine.
  (5) Mika's order asks for proof of age.
  (6) Mika does not have a proof of age.
  (7) Mika's order returns False.
  (8) Mika adds juice to her order.
  (9) Mika gives order to Jane.

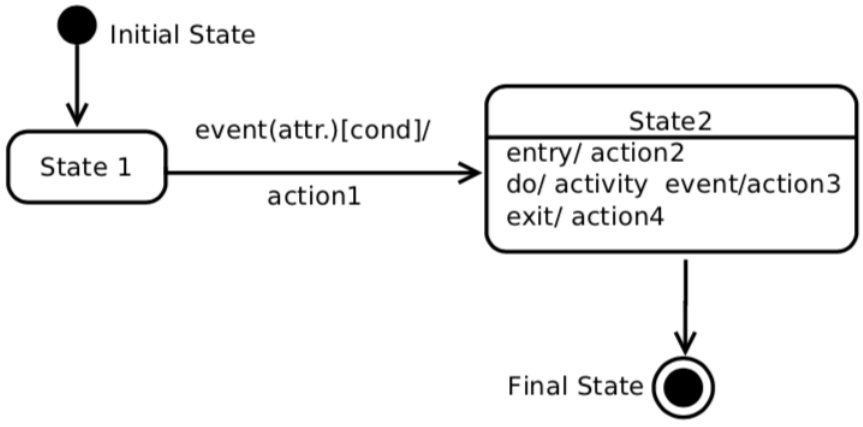
- **Activity Diagram**
- Activity diagrams are graphical representations of workflows (or flowcharts) of stepwise activities and actions, with support for choice, iteration, and concurrency.
- In an activity diagram,
  ellipses represent actions;
  diamonds represent decisions;
  a black circle represents the start (initial node);
  an encircled black circle represents the end (final node);
  bars represent the start (split) or end (join) of concurrent activities;
  rectangles represent objects, classes.
- Here is an activity diagram to find the gcd of two positive integers using the inefficient guessing approach:



- Draw an activity diagram to find the gcd of two positive integers using Euclid's algorithm.

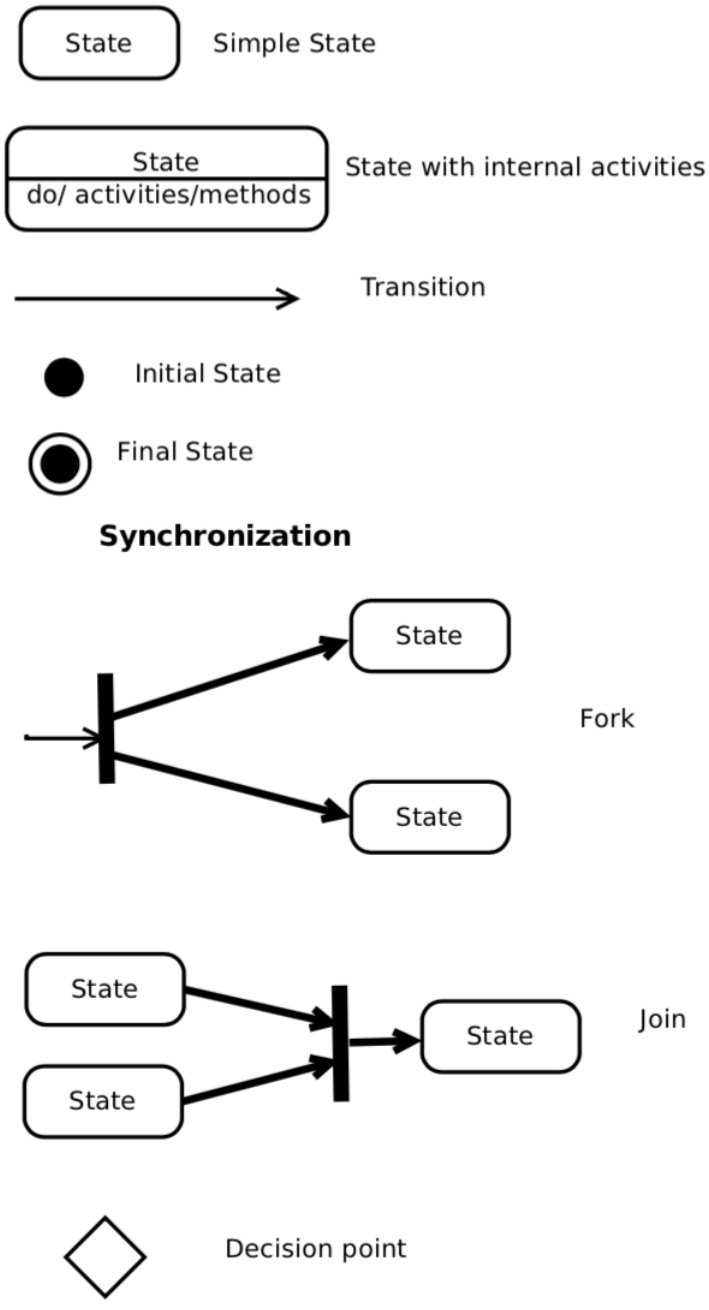
- **State Diagram**
- A state diagram shows the behavior of classes in response to external stimuli.
- A state diagram describes the behavior of a single object in response to a series of events in a system.
- A state diagram is also sometimes known as State Chart, or State Machine, or State Transition Diagram.
- In general, state diagrams are studied for an entire system, or a subsystem or one particular object.
- A state diagram has three important building blocks: State, Event (which triggers an action or a transition), Transition.
- The State is the status of the system or object at the given time. States may also have activities or actions within themselves.
- The Event is the trigger which causes the state to change. This event could be internal or external.
- Transition is the change in state.
- **Notations:**
- **Initial State:** This is denoted by a solid filled in circle. Initial state is a pseudo-state. Its purpose is to signify the position from where the diagram starts. There may or may not be a transition to transition from this initial state to the first system state.
- **State:** This is represented by a rectangle with smoothened vertices (rounded corners). The name of the state is written inside, on the top. For instance, an activity is an operation that the object has to perform when it is in a given state.
- **Transition:** This is represented by a simple arrow, starting from the source state and pointed towards the transitioned new state. Each transition is labeled by an event or a trigger resulting in an action. The notation used here is *Event/Action*. Sometimes, there is a *guard* condition. This guard condition controls whether the action is performed or not. The condition is written in brackets [ ].
- **Final State:** This is denoted by a solid filled in circle with a concentric circle around it. Transitioning to this state represents the completion of the state diagram.

State Transition Diagram: Notations



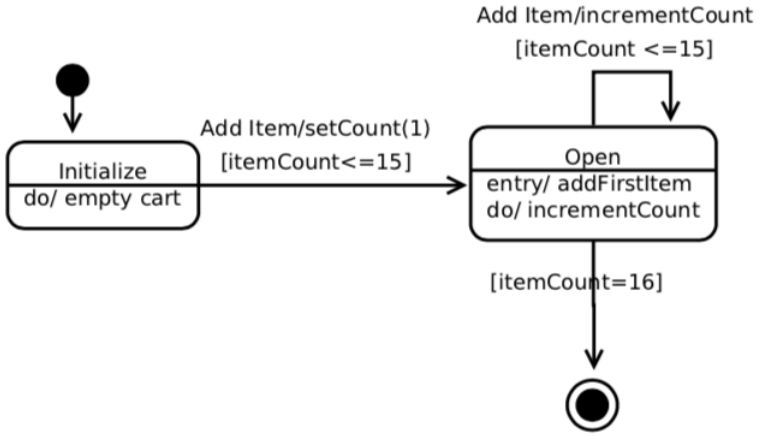
- State diagrams are often confused with activity diagrams or flowcharts. Note, state machine performs actions in response to events. Flowcharts do not need explicit events. Secondly, a flowchart illustrates the **processes** that are executed in the system. A state diagram shows the **actual changes** in state (not the processes or commands that created those changes).

More Notations:



- Note, all synchronization bars must be paired, so that all the forks eventually join and come back to one transition.

- Here is a short example: State Transition Diagram for a Shopping Cart.



- Here is an example that you could try:
- Processing Order:
- The first state of the order is “Unpaid”.
- We go through a Decision Point which allows us three options: “100% Cash paid”, “100 % Card paid”, a fork (going through “50% Cash paid”, and “50% Card paid”).
- The three above options (fork goes through synchronization bars to join) lead to “Paid” state.
- Do not forget the initial and final states.

3. Built-in Python classes (list, str) and numeric types (int, long, float) (sections 2.2, 2.3, 2.4, 2.5)

- We recall a built-in Python class, the list.
- An empty list is constructed using the constructor of the *list* class. Consider the following **assignment** statement:

```
groceries = list()
```

- Notice that the list you have created is an empty list. The construction of a new object from a given class is called **instantiation**.
- The word *groceries* is the **identifier** – it is the name. An identifier must consist of letters, digits, or underscore characters, yet a digit cannot be the first character of an identifier.
- The identifiers are case sensitive. Further, keywords of Python cannot be used as identifiers.
- An assignment statement is used to assign a label to an object. Here is the result of the assignment statement:

FIGURE 2.1: The steps in the execution of the groceries = **list**( ) statement: (a) creation of an empty **list** instance, and (b) the assignment of identifier groceries to the newly created object.

- To see how our list looks like, type in

```
groceries
```

- Now we can start populating our list, *groceries* using the *append* method. Note how an object method is called:

```
groceries.append('bread')
groceries.append('milk')
groceries.append('cheese')
```

- Now see how our list *groceries* looks like.
- What happens if you try again

```
groceries.append('bread')
```

- Note that there is a more convenient, a **literal** form to instantiate a list as:

```
groceries2=['bread','milk','cheese','bread']
```

- Now build any list(s) you like, and implement **every** method given on the next page:

Behaviors that modify an existing list (i.e., mutators)	
data.append(val)	Appends val to the end of the list.
data.insert(i, val)	Inserts val following the first i elements of the list.
data.extend(otherlist)	Adds the contents of otherlist to the end of this list.
data.remove(val)	Removes the earliest occurrence of val found in the list.
data.pop( )	Removes and returns the last element of the list.
data.pop(i)	Removes and returns the element with index i.
data[i] = val	Replaces the element at index i with given val.
data.reverse( )	Reverses the order of the list's elements.
data.sort( )	Sorts the list into increasing order.

Behaviors that return information about an existing list (i.e., accessors)	
len(data)	Returns the current length of the list.
data[i]	Returns the element at index i.
val **in** data	Returns **True** if the list contains val, **False** otherwise.
data.count(val)	Counts the number of occurrences of val in the list.
data.index(val)	Returns the index of the earliest occurrence of val.
data.index(val, start)	Returns the index of the earliest occurrence of val that can be found starting at index start.
data.index(val, start, stop)	Returns the index of the earliest occurrence of val that can be found starting at index start, yet prior to stop.
dataA == dataB	Returns **True** if contents are pairwise identical, **False** otherwise.
dataA != dataB	Returns **True** if contents *not* pairwise identical, **False** otherwise.
dataA < dataB	Returns **True** if dataA is lexicographically less than dataB, **False** otherwise.

Behaviors that generate a new list as a result	
data[start **:** stop]	Returns a new list that is a "slice" of the original, including elements from index start, up to but not including index stop.
data[start **:** stop **:** step]	Returns a new list that is a "slice" of the original, including elements from index start, taking steps of the indicated size, stopping *before* reaching or passing index stop.
dataA + dataB	Generates a third list that includes all elements of dataA followed by all elements of dataB.
data * k	Generates a new list equivalent to *k* consecutive copies of data (i.e., data + data + ... + data).

FIGURE 2.2: Selected **list** behaviors, for prototypical instances data, dataA, and dataB.

- Try the following and see what happens (explain the result):

```
mylist = [3,2,14,5,6,2]
mylist = mylist.sort()
mylist
```

- Try the following and see what happens (explain the result):

```
groceries = ['milk', 'bread', 'cereal']
groceries = groceries.sort()
groceries
```

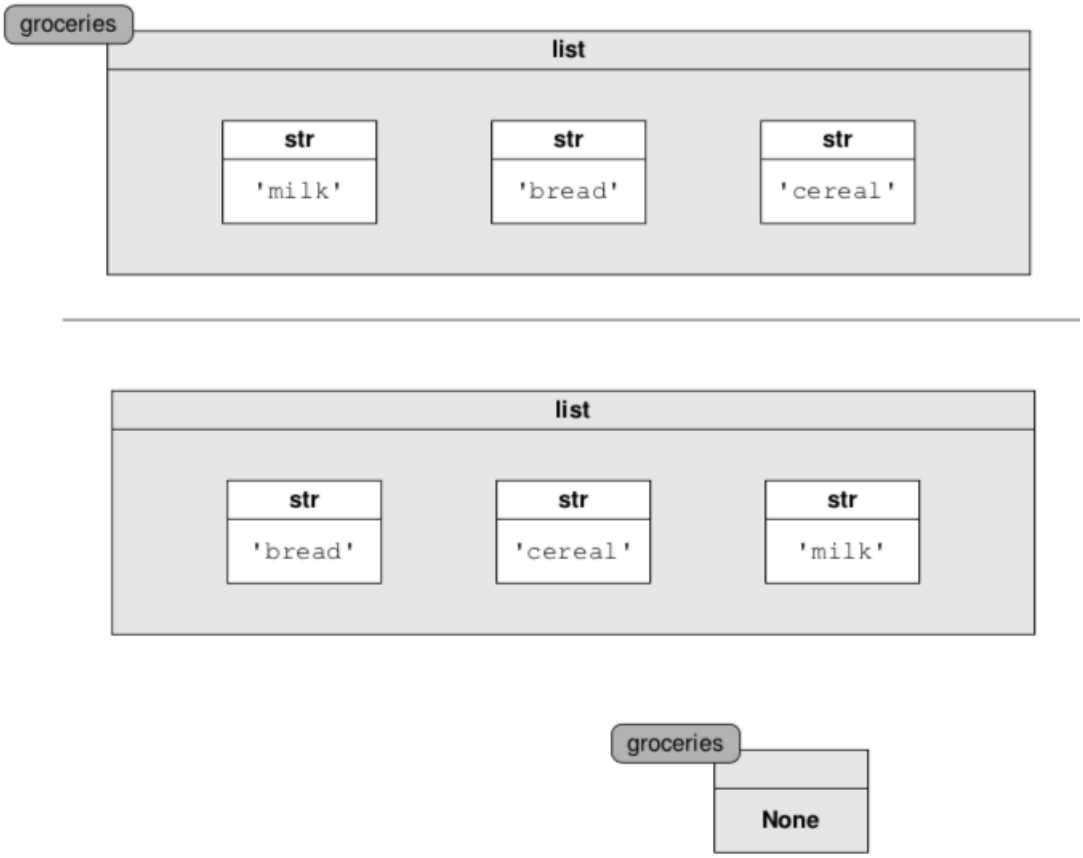


FIGURE 2.3: A before-and-after view of the command groceries = groceries.sort( ).

- Please watch out for such unexpected results when you go through the list-methods given on the previous page.

- **Other Sequence Classes: Strings and Tuples.**
- Notice, lists are **mutable** objects. They can change *in place.*
- Examples of **immutable** classes: **str** class (the string class), and the **tuple** class.
- The **str** class is designed for sequences of characters.
- A **tuple** class is an immutable version of the **list** class. It represents a sequence of arbitrary objects, yet a sequence that cannot be altered.
- Although **list**, **str**, and **tuple** are three distinct classes, they support many of the same operations. For instance, data[i] retrieves the element at index i.
- Such a generic syntax is called **polymorphic**.
- The constructor for the string class is invoked as str() which creates an empty string. Note, a string is immutable. So we cannot populate a string.
- So, we create a string object using literal form. For example:

```
knockknockjoke = "Knock Knock \n Who's there? \n Banana ..."
```

- Do you see why double-quotes were used in the example above?
- Strings can be compared to each other; keep in mind that strings are case sensitive.
- 'Hello' == 'hello' will return False.
- String inequalities are $<, >, <=, >=$ based on lexicographical order. Note, though uppercase letters come before smaller case letters. Thus, 'Z' < 'a'.
- On the next couple of pages we list several string methods. Before we do that, what do you think is the outcome of the following? Why?

```
person = 'Alice'
person = person.lower()
person
```

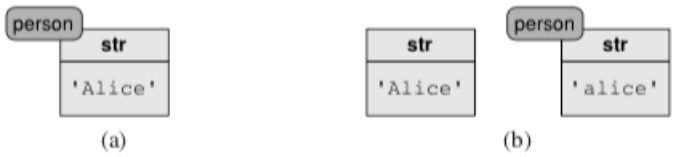


FIGURE 2.4: A before-and-after look at the command person = person.lower( ).

- Now construct a few strings, and implement **every** method from the next two pages:

Behaviors that return information about the existing string s	
len(s)	Returns the current length of the string.
s[i]	Returns the character at index i.
pattern **in** s	Returns **True** if the given pattern occurs as substring of s, **False** otherwise.
s.count(pattern)	Returns the number of distinct occurrences of pattern within s.
s.find(pattern)	Returns the index starting the leftmost occurrence of pattern within s; if pattern is not found, returns −1.
s.find(pattern, start)	Returns the index starting the leftmost occurrence of pattern found at or after index start within s; if no such pattern is found, returns −1.
s.rfind(pattern)	Returns the index starting the rightmost occurrence of pattern within s; if pattern is not found, returns −1.
s.rfind(pattern, start)	Returns the index starting the rightmost occurrence of pattern found at or after index start within s; if no such pattern is found, returns −1.
s.index(pattern)	Same as s.find(pattern) except causes ValueError if not found.
s.index(pattern, start)	Same as s.find(pattern, start) except causes ValueError if not found.
s.rindex(pattern)	Same as s.rfind(pattern) except causes ValueError if not found.
s.rindex(pattern, start)	Same as s.rfind(pattern, start) except causes ValueError if not found.
s.startswith(pattern)	Returns **True** if s starts with the pattern, **False** otherwise.
s.endswith(pattern)	Returns **True** if s ends with the pattern, **False** otherwise.
s.isalpha( )	Returns **True** if all characters are alphabetic, **False** otherwise.
s.isdigit( )	Returns **True** if all characters are digits, **False** otherwise.
s.isspace( )	Returns **True** if all characters are whitespace, **False** otherwise.
s.islower( )	Returns **True** if all alphabetic characters are lowercase, **False** otherwise.
s.isupper( )	Returns **True** if all alphabetic characters are uppercase, **False** otherwise.
s == t	Returns **True** if strings are identical, **False** otherwise.
s != t	Returns **True** if strings are *not* identical, **False** otherwise.
s < t	Returns **True** if string s is alphabetized before t, **False** otherwise.

FIGURE 2.5: Selected behaviors supported by Python's **str** class (continued on next page).

Behaviors that generate a new string as a result	
s.capitalize( )	Returns a capitalized version of the original string.
s.lower( )	Returns an entirely lowercase version of original.
s.upper( )	Returns an entirely uppercase version of original.
s.center(width)	Returns a new string of the specified width containing characters of the original centered within it.
s.ljust(width)	Returns a new string of the specified width containing characters of the original left justified within it.
s.rjust(width)	Returns a new string of the specified width containing characters of the original right justified within it.
s.replace(old, new)	Returns a copy of s, with every occurrence of the substring old replaced with new.
s.strip( )	Returns a copy of s with leading and trailing whitespace removed.
s.strip(chars)	Returns a copy of s with as many leading and trailing characters from the string chars removed.
s[start:stop]	Returns a new string that is a "slice" of the original, including characters of original from index start, up to but not including index stop.
s[start:stop:step]	Returns a new string that is a "slice" of the original, including characters of original from index start, taking steps of of the indicated size, continuing up to but not including index stop.
s + t	Generates a third string that is the concatenation of the characters of s followed by the characters of t.
s * k	Generates a new string equivalent to *k* consecutive copies of s (equivalent to s + s + ... + s).

Behaviors that convert between strings and lists of strings	
s.split( )	Returns a **list** of strings obtained by splitting s into pieces that were separated by whitespace.
s.split(sep)	Returns a **list** of strings obtained by splitting s into pieces that were separated by sep.
s.join(stringSeq)	Returns a string that is a concatenation of all elements of stringSeq with a copy of s inserted between each pair.

FIGURE 2.5 (continuation): Selected behaviors supported by Python's **str** class.

- **Tuples:**
- A tuple is an immutable sequence of objects.
- Because tuples are immutable, Python can implement them more efficiently than the corresponding lists and need only support a subset of the behaviors afforded to lists.
- The literal form for tuples uses enclosing parentheses, with commas separating individual elements:

```
person = ('Firstname', 'Lastname', 'Profession')
```

- Tuples support all the nonmutative behaviors of lists that wee summarized in Figure 2.2 with the exception of count and index.
- Construct a few tuples and implement **every** tuple method from Figure 2.2
- **Numeric Types: int, long, and float:**
- There are four different primitive types for storing numbers: **int, long, float,** and **complex**.
- There is a maximum magnitude for a value stored as an int, although this limit depends upon the underlying computer architecture.
- For integer values that have maginitude beyond that limit, Python supports a long class that can perfectly represent integers with arbitrarily large magnitudes (although with a more complicated internal encoding).
- Python automatically converts from the use of into long when necessary.
- Floating-point representation is used for real numbers which are integers. In Python, these are stored using a class named float. Note, the literal 3.0 belongs to the float class even though it has not fractional part
- We will not work much with complex numbers.
- Here are selected operators supported by Python's numeric types:

Syntax	Semantics
x + y	Returns the sum of the two numbers.
x − y	Returns the difference between the two numbers.
x * y	Returns the product of the two numbers.
x / y	Returns the result of a true division[a] of x divided by y.
x // y	Returns the integral quotient of x divided by y.
x % y	Returns the remainder of an integer division of x divided by y.
x ** y	Returns x^y; also available as pow(x,y).
−x	Returns the negated number.
abs(x)	Returns the absolute value of the number.
x == y	Returns **True** if x and y are equal, **False** otherwise.
x != y	Returns **True** if x is not equal to y, **False** otherwise.
x < y	Returns **True** if x is less than y, **False** otherwise.
x <= y	Returns **True** if x is less than or equal to y, **False** otherwise.
x > y	Returns **True** if x is greater than y, **False** otherwise.
x >= y	Returns **True** if x is greater than or equal to y, **False** otherwise.
cmp(x, y)	Returns −1 if x < y, 0 if x = y, and 1 if x > y.

FIGURE 2.6: Selected operators supported by Python's numeric types.

- Note, // represents integer division and % represents the modulo operator.

- These numeric types are immutable. A typical command $age = age + 1$ does not alter the value of the original object; it creates a new object.

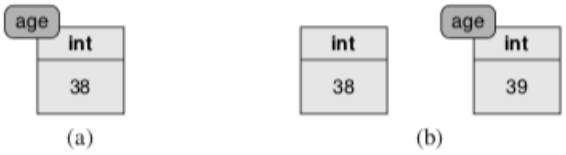


FIGURE 2.7: A before-and-after look at the command age = age + 1.

- **Type Conversions:**
- Converting information from one data type to another is called **casting**.
- Try the following conversion commands appropriately:
- **int, float, round, str**
- A second notion of conversion between text and numbers involves the low-level encoding of text characters.
- Each character is stored in memory as an integer, using ASCII (American Standard Code for Information Interchange which is a 7-bit character set for 128 characters) or Unicode (a much larger character set – www.unicode.org).
- The function ord(character) returns the integer code for the chosen character.
- The function chr(integer) converts the code value to the corresponding character.
- Remember 'Z' < 'a'. Find their respective ordinals.
- We can convert a string to a list of characters using *list(string)*.
- Note, the parameter in the list() constructor can be another list or a tuple.
- Likewise, the tuple() constructor can be used to create a tuple based on the contents of an existing tuple, list or string.
- Try these various conversions using concrete examples.

4. Expressions, Calling Functions (sections 2.6, 2.7, 2.8)

- We have seen member functions of a class:

```
object.method(parameters)
#For example:
circle.draw(window)
math.sqrt(25)
```

- We have also seen **pure functions**; that is, functions which are not formally methods from a class.

```
function(parameters)
#For example:
chr(37)
ord(" ")
int(3.14)
```

- Here is a table of some commonly used built-in function. Try **every one** of these with various parameters (think numeric, string, lists, tuples types).

Sample Syntax	Semantics
range(stop)	Returns list of integers from 0 up to but not including stop.
range(start, stop)	Returns list of integers from start up to but not including stop.
range(start, stop, step)	Returns list of integers from start up to but not including stop.
len(sequence)	Returns the number of elements of the sequence.
abs(number)	Returns the absolute value of the number.
pow(x, y)	Returns x^y.
round(number)	Returns a **float** with integral value closest to the number.
ord(char)	Returns the integer alphabet code for the given character.
chr(code)	Returns the character having the given integer alphabet code.
min(a, b, c, ...)	Returns the "smallest" of the given parameters.
min(sequence)	Returns the "smallest" item in the sequence.
max(a, b, c, ...)	Returns the "largest" of the given parameters.
max(sequence)	Returns the "largest" item in the sequence.
sum(sequence)	Returns the sum of the given sequence of numbers.
sorted(sequence)	Returns a copy of the given sequence that is sorted.[a]

FIGURE 2.8: Several commonly used built-in functions.

[a] Introduced in version 2.4 of Python.

- **Python Modules:**
- There are hundreds of useful tools other than built-in functions which have been written by developers of Python.
- These tools are placed into libraries, called **modules**, that can be individually loaded as needed.
- In order to use the tools from a particular library, we need to **import** that library.

```
import library
library.function(parameters)
#For example:
import math
math.sqrt(25)
import random
random.random()
```

- You may also selectively import a few functions or all the functions from a library:

```
from library import function1, function2
function1(parameters)
function2(parameters)
from library2 import * #Import all (except internal) functions

#For example:
from math import *
sqrt(25)
pi
from random import randrange
randrange(2,25)
```

- We can obtain information regarding a library with the command *help(library)* after importing the said library. Here are some commonly used libraries.

Name	Overview
math	Various mathematical constants and functions (e.g., pi, sqrt, sin, cos, tan, log).
random	Classes and functions used to support randomization of data according to various distributions (e.g., randint, sample, shuffle, uniform).
time	Various functions to manipulate time (e.g., sleep, time).
datetime	Classes and function to manipulate dates and times (e.g., time, date, datetime classes).
re	Support for string matching with ***regular expressions***.
os	Support for interactions with the operating system.
sys	Values and functions involving the Python interpreter itself.

FIGURE 2.9: Several commonly used modules.

- Here are some concrete examples:

```
import math
math.sqrt(25)
math.pi
math.sin(math.pi /2)
math.tan(3*math.pi/4)
math.factorial(6)
math.floor(5.239)
math.ceil(5.239)
#######################
import random
random.random()
random.randrange(4,20,3)
random.uniform(4,20)
#########################
import time
time.time()  #Number of seconds since 12:00 am, Jan1, 1970 (epoch)
time.localtime(time.time()) #Structured local time

print('Hello')
time.sleep(5)
print('Good morning')
#############################
import datetime
datetime.datetime.now() #..... microseconds?
datetime.date.today()
#############################
import re
xx='Hello, how are you?'
yy='May I have some water?'
zz='gibberish.blah'
re.findall(r'^\w+',xx)
re.findall(r'^\w+',yy)
re.findall(r'^\w+',zz)
#######################################
import os
os.getcwd()
os.chdir("c:\Users\(username)\Desktop")
os.path.expanduser('~/Desktop/')
```

- **Expressions:**
- We have seen expressions such as $14 + 5 + 3$, or $14 + 15 - 2$, or $14 - 15 + 2$.
- You need to know order of operations or **precedence** to evaluate expressions.
- Note, addition and subtraction have equal precedence, and are **left-associative**. Likewise, multiplication and division have equal precedence, and are left-associative. On the other hand, exponentiation is right-associative. That is, $4 * *3 * *2 = 4^{3^2} = 4 * *(3 * *2) = 4^{(3^2)}$.
- We portray the evaluation order graphically using a hierarchical **evaluation tree**. For example:

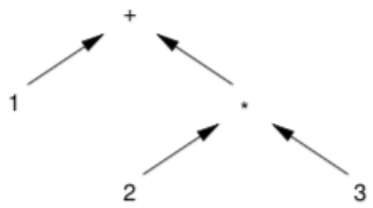


FIGURE 2.10: An evaluation tree demonstrating the order of operations for the expression 1 + 2 * 3.

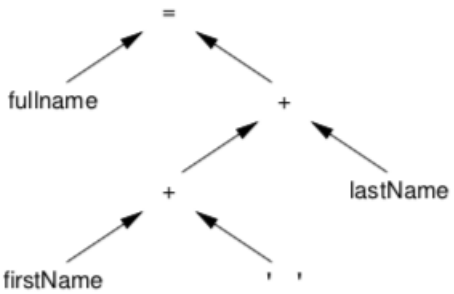


FIGURE 2.11: An evaluation tree demonstrating the order of operations for the expression fullName = firstName + ' ' + lastName.

- Note that the **assignment operator** (=) has the *lowest* precedence. This is why
  fullName = firstName + ' ' + lastName is equivalent to
  fullName = ((firstName + ' ') + lastName).
- **Boolean Expressions and the bool Class:**
- The literals **True** and **False** are instances of the class **bool**.
- The term bool is named after the mathematician George Boole who pioneered the study of many logical properties.
- There are three core logical operators: **not** (this is unary), **(**and) (binary), **or** (binary).

- Here is the truth table for these boolean operators:

x	y	not x	x and y	x or y	x == y	x != y
False	False	True	False	False	True	False
False	True	True	False	True	False	True
True	False	False	False	True	False	True
True	True	False	True	True	True	False

FIGURE 2.12: Truth table for the common boolean operators. Given the values of x and y shown in the left two columns, the remaining columns show the resulting value of subsequent expressions.

- Note that the *or* is an **inclusive or**.
- The **exclusive or** relies on $x! = y$ (that is, both x and y cannot simultaneously be True).
- Some compound boolean expressions can be **chained:** For example,
  $3 < x$ and $x < 8$ is equivalent to $3 < x < 8$.
- **Calling Functions from Within Expressions:**
- Here is an example: $len(groceries) > 15$ and 'milk' in groceries
- Here is an evaluation tree for this expression:

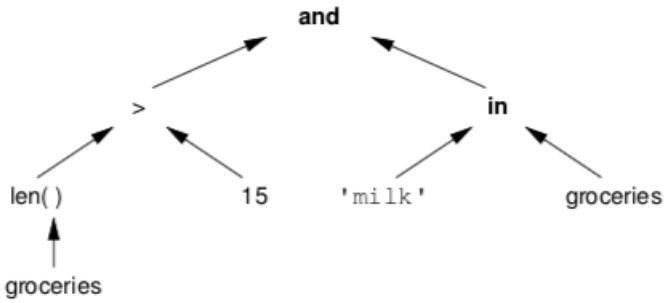


FIGURE 2.13: An evaluation tree demonstrating the order of operations for the expression len(groceries) > 15 **and** `'milk'` **in** groceries.

- Note that when multiple functions calls are used in the same expression, they are typically evaluated from left to right. For example, consider the following and give their output:

```
song = 'With a Moo Moo Here And a Moo Moo There'
song.split()[5]

song.lower().count('t')

myList = song.lower().split()
myList.insert(myList.index('moo') +2, 'quack')
myList
```

- The order in which commands are executed by a program is its **flow of control.**
- By default, statements are executed in the order in which they are given.
- A different execution order can be specified using **control structures.**
- The **for loop** is one such control structure.
- The repetition of a series of steps for each item of a sequence is called **iteration**.
- A for loop always begins with the syntax *for identifier in sequence:* followed by a block of code we call the *body* of the loop.

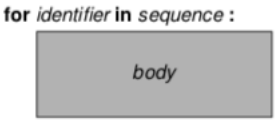


FIGURE 4.1: General form of a for loop.

```
for person in guests:
  print person
```

- Note, in the above example, *person* is the **loop variable**. The loop variable name is expected to be meaningful.
- The sequence in the above example is *guests*. It can be a sequence of elements, usually a list, string, or tuple.
- The first line ends with a colon (:) to designate the forthcoming body.
- The body is indented.
- Here is a concrete example and its diagram:

```
guests = ['Carol', 'Alice', 'Bob']
for person in guests:
    print('Hello my name is', person)
```

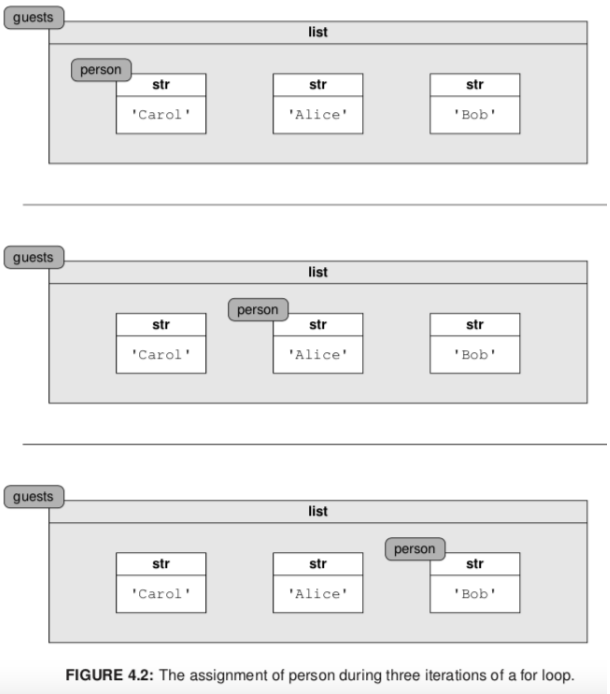
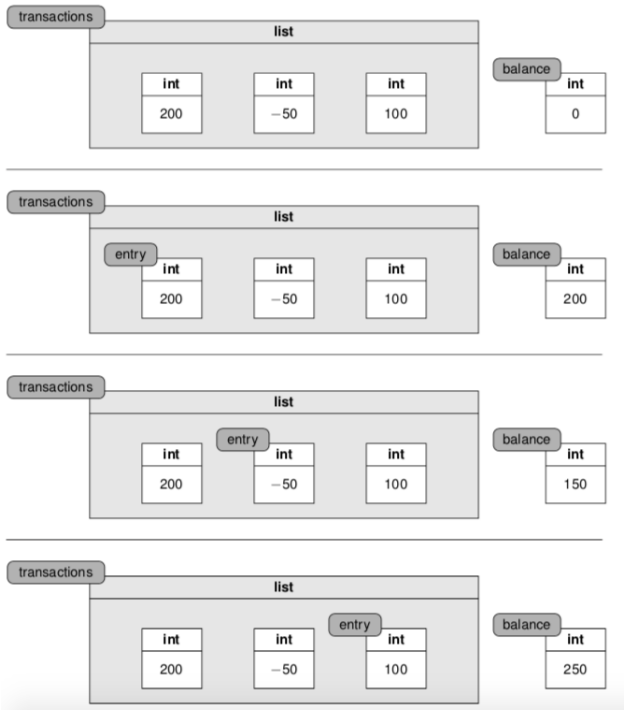


FIGURE 4.2: The assignment of person during three iterations of a for loop.

- A for loop can technically iterate upon an empty sequence, the body of the loop is never executed; there are no elements.
- Here is another example:

```
balance = 0
transactions = [200,-50,100]
for entry in transactions:
    balance = balance + entry. #Can also be written,  balance += entry
print('Your balance is', balance) #Not indented
```



- Since the print statement above is not indented, it is not part of the body of the for loop. In other words, the print statement is implemented after the for loop is completed.
- Note that we can use the for loop in an interactive session with the interpreter also.
- To get out of the for loop, we need to hit 'Enter' twice.
- What is the output of the following code?

```
for count in range(10,0,-1):
    print(count)
print('Blastoff!')
```

- What is the output of the following code? (This technique is **index-based loop**)

```
groceries = ['milk', 'cheese', 'bread' , 'cereal']
for position in range(len(groceries)):
    label = str(1+position)+'. '
    print(label+groceries[position])
```

- What is the output of the following code?

```
guests = ['Carol', 'Alice', 'Bob']
for person in guests:
    person = person.lower()
guests
```

- Now compare with the output of the following code:

```
guests = ['Carol', 'Alice', 'Bob']
for i in range(len(guests)):
    guests[i] = guests[i].lower()
guests
```

- **Nested Loops:**
- The technique of using one control structure within the body of another is called nesting. For example,

```
for chapter in ('1','2'): #Outer Loop
    print('Chapter '+chapter)
    for section in ('a','b','c'):  #Inner Loop
        print('Section "+chapter+section)
print('Appendix')
```

- **List Comprehension:**
- Consider the following piece of a code:

```
auxiliary = []
for person in guests:
    auxiliary.append(person.lower())
auxialary
```

- This is an accumulator approach. We start with an empty list, and then populate it using a for loop.
- Python supports a simpler syntax for such tasks known as **list comprehension**.

```
auxiliary = []
auxiliary = [person.lower() for person in guests]
auxialary
```

- In general, list comprehension follows the syntax

```
result = [expression for identifier in sequence]
```

- In fact, there is a more general form of list comprehension using condition

```
result = [expression for identifier in sequence if condition]
```

- For example,

```
deposits = [entry for entry in transactions if entry > 0]
```

6. While loops, defining functions (sections 5.1, 5.2, 5.4)

- The **while loop** is a control structure which is used when a programmer needs greater flexibility than is offered by the for loop.
- The for loop requires a pre-existing sequence of objects. The while loop is used when such a sequence is not available in advance.
- The general syntax for the while loop is

```
while condition:
    body
```

- For example, in an interactive game, we could have the following code

```
response = input("Shall we play a game? ")

while response.lower() in ('y','yes'):
    #...code for playing a game
    response = input("Would you like to play again? ")
```

- Here is the flow chart for the while loop:

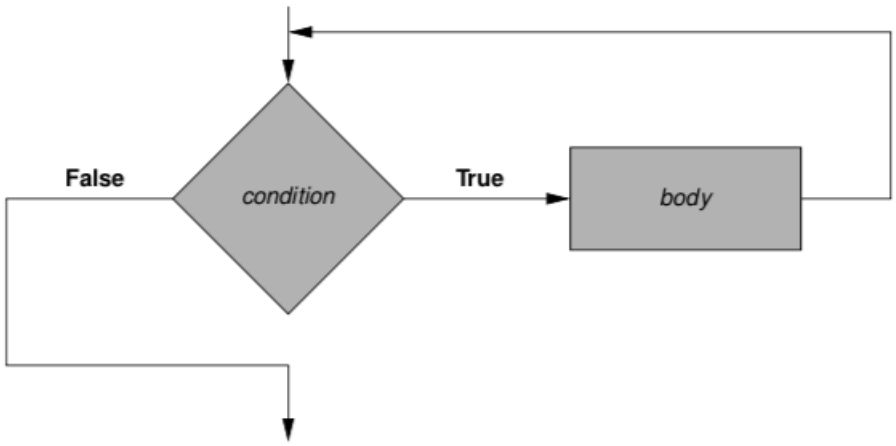


FIGURE 5.1: Semantics of a while loop.

- Here is an algorithm to find the GCD of two positive integers.

```
u = int(input("Enter first positive integer: "))
v = int(input("Enter second positive integer: "))

guess = min(u, v)
while (u % guess >0) or (v % guess >0):
    guess -= 1.

print("The gcd is", guess)
```

- The following is the more efficient Euclid's algorithm:

```
u = int(input("Enter first positive integer: "))
v = int(input("Enter second positive integer: "))

while (v != 0):
    u,v = v, u % r

print("The gcd is", u)
```

- In the following code, a non-empty string is interpreted as the boolean value True:

```
guests = []
name = input("Enter a name (blank to end): ")
while name:
    guests.append(name)
    name = input("Enter a name (blank to end): ")
print(f"You entered {len(guests)} guests.")
```

- The loop above is a post-test loop. It is also a loop and a half. We can avoid repeating the command for reading a name with the following pre-test loop (that is, the body of the loop is forced to be executed at least once):

```
guests = []
name = 'fake'
while name:
    name = input("Enter a name (blank to end): ")
    if name:
        guests.append(name)
print(f"You entered {len(guests)} guests.")
```

- In the loop above, while we have avoided repeating the command for reading a name, we have repeated the check whether the name is blank. The following code removes the extra blank entry at the end:

```
guests = []
name = 'fake'
while name:
    name = input("Enter a name (blank to end): ")
     guests.append(name)
guests.pop()
print(f"You entered {len(guests)} guests.")
```

- Here is another example:

```
number = 0
while not (1 <= number <= 10):
    number = int(input("Enter a number from 1 to 10:  "))
    if not (1 <= number <= 10):
        print("Your number must be from 1 to 10.  ")
```

- **Infinite Loops:**
- Here is an example of an infinite loop:

```
while True:
    print("Hello")
```

- Here are pieces of code one of which involves an infinite loop:

```
i=0
while i < 11:
  print(i, i**2)


#Here is the second one
i=0
while i < 11:
  print(i, i**2)
  i += 1
```

- To manually force the interpreter to stop the execution of an infinite loop is typically done by entering the control-c key stroke.
- **Functions:**
- Functions can serve as the ultimate control structure.
- A function allows a series of complicated instructions to be encapsulated and then subsequently used as a single high-level operation.
- For example, suppose we want to design and implement a function that locates the maximum-length string from a sequence of strings. Say, we want to implement this function in the following fashion:

```
ingredients = ['carbonated water', 'caramel color',
        'phosphoric acid', 'sodium saccharin', 'potassium benzoate',
         'natural flavors', 'citric acid', 'caffeine',
         'potassium citrate', 'aspartame', 'dimethylpolysiloxane']
concern = maxLength(ingredients) #calling our function
print(concern)
```

- Note that *maxLength()* is not a built-in pure Python function, nor is a list method. So, we define this function:

```
def maxLength(stringSeq):
    longSoFar = ''   #the longest string so far is the empty string
    for entry in stringSeq:
        if len(entry) > len(longSoFar):
            longSoFar = entry
    return longSoFar
```

- Note the syntax for defining functions. We start with **def functionName(parameters)**. In our case, there is only one parameter, a list or tuple of strings. Notice also that this function has to **return** something (in our case, the entry with the maximum length). If the definition of a function does not have a return statement, then it returns an object *None*.

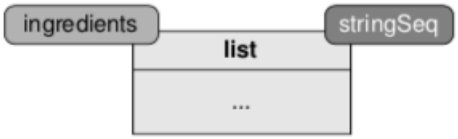


FIGURE 5.2: The parameter is identified as ingredients from the context of the caller, yet as stringSeq within the context of the function body.

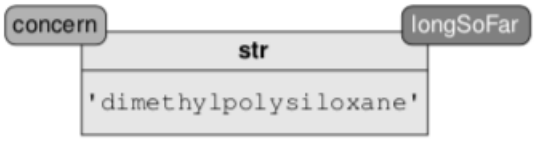


FIGURE 5.3: The mechanism for passing a return value is similar to that for a parameter. Upon return, the caller's identifier concern is associated with the object identified internally as longSoFar.

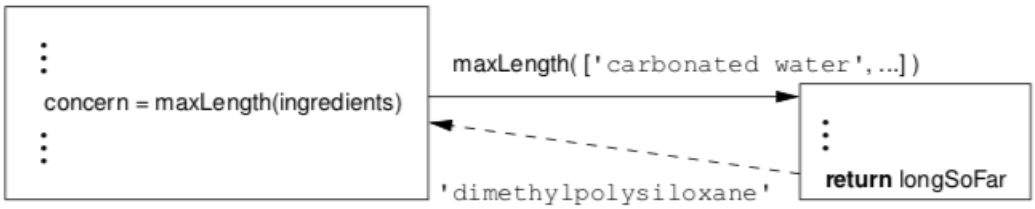


FIGURE 5.4: Flow of control for a function call.

- **Parameters:** Note, in our case, *stringSeq* is the **formal parameter** while *ingredients* is the **actual parameter**. Sometimes a function does not take any parameters. In that case we use empty parentheses () in the definition and calling of the said function.
- Each time a function is called, the system assigns the identifier we chose as a formal parameter to the actual parameter indicated by the caller. The code in the body of the function uses the formal parameter to identify the underlying piece of information.
- **Body:** The body of the function is indented after the **def** statement. In our example, the identifier *longSoFar* has **local scope** since it cannot be directly accessed by any block of code other than the body of the function; it is created solely for our use in processing the

current function call. In other words, if we have another *longSoFar* outside the definition of the function *maxLength*, then the two variables do not interfere.

- **Return value:** The function body ends with **return longSoFar**. Although the local identifier longSoFar will cease to exist as the function ends, the underlying object will be available to the caller.
- Sometimes the *return* command is executed elsewhere within the body of the function (for example, within a conditional). In this case, the execution of the function immediately ends with any specified value passed back to the caller. Sometimes we may use a return statement without any subsequent value. In this case, the execution of the function without returning any formal value to the caller, and the special value **None** object is returned.
- **Flow of Control:** Calling a function is a form of a control statement. Parameters are sent to the function and the moment the caller invokes the function, control is passed to the body of the function. Some function calls may take more time to complete than others. Sometimes, a function may get stuck in an infinite loop. The control returns to the caller after the function is completed.
- More generally, we can nest function calls just as we nest other control structures. For instance, consider the following diagram of the flow of control:

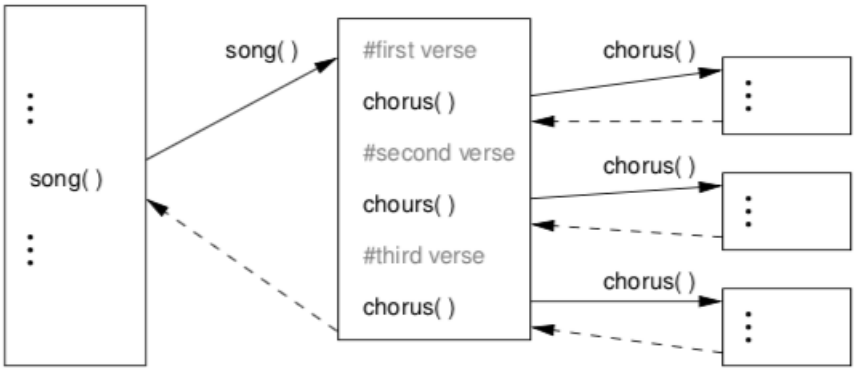


FIGURE 5.5: Flow of control for nested function calls.

- **Optional Parameters:** Sometimes we can define functions with optional parameters. In this case, a caller can choose to send a value for such a parameter, but otherwise a default value is substituted. For example,

```
def countdown(start=10, end =1):
    for count in range(start, end-1, -1):
      print(count)
```

- The function *countdown* can be called in the following ways

```
countdown(30, 5) #clearly stating the parameters;
countdown() #here the default parameters of 10,1 are used;
countdown(50) #here, the first parameter is 50
          while the second parameter is the default of 1.
```

- It is also possible to define functions for which some parameters have default values and others do not. Build various examples.

- **Case Study: Computing the Square Root**
- While the math library has a sqrt() function, we will try Newton's method to define our own sqrt() function. We go through several improvements in the process.
- We start with a *guess* square-root.

  Note, *guess * guess = number* is equivalent to *guess = number/guess*.
- Here is a first attempt:

```
def sqrtA(number):
    guess = 1.0
    while guess != number/guess:
        guess = (guess + number/guess)/2.0
                        #improving our guess by using average
    return guess
```

- This code will get into an infinite loop because we can keep improving our guess, since most square-roots are irrational.
- Here is an improvement using **fixed number of iterations**:

```
def sqrtB(number):
    guess = 1.0
    for trial in range(100):
                #we are going to improve our guess 100 times
        guess = (guess + number/guess)/2.0
                 #improving our guess by using average
    return guess
```

- While this approach works better, sometimes an exact square root is found within a few iterations, then the program continues for the rest of the iterations unnecessarily.
- So here is another approach where the iterations continue until the gap is small (as in an allowableError):

```
def sqrtC(number, allowableError = 0.000001):
    guess = 1.0
    while abs(guess - number/guess) > allowableError:
        guess = (guess + number/guess)/2.0
                 #improving our guess by using average
    return guess
```

- This function can get into an infinite loop if the caller passed 0 as an allowableError. Even in the default value, we can end up in an infinite loop.

- Let us try one more option:

```
def sqrtD(number):
    prevGuess = 0.0
    guess = min(1.0, number)
    while prevGuess < guess:
        prevGuess = guess
        avg = (guess + number/guess)/2.0
        guess = min(avg, number/avg)
    return guess
```

7. Conditional Statements (section 4.4)

- **Conditional statement**, or more commonly, an **if statement** is another control structure.
- The syntax for an **if statement** is:

```
if condtion:
    body
```

- The flow chart for an if statement is given:

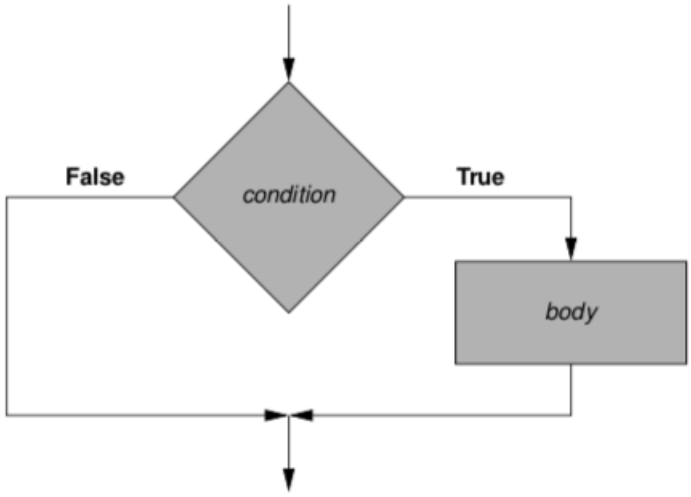


FIGURE 4.11: Flowchart for an **if** statement.

- For example:

```
dinner = input("What would you like for dinner?")
if dinner == 'pizza':
    print("Great!")
    print("I love pepperoni and black olives.")
```

- **The condition:** Note, the condition can be a boolean expression. For example:

```
if len(groceries)>15 or 'milk' in groceries:
    print("Go to the grocery store")

#Here is another example -- see the difference?
if len(groceries)>15 and 'milk' in groceries:
    print("Go to the grocery store")
```

- **The body:** The body of the loop is indented after the if statement. The body could contain another conditional statement, resulting in nested if-statements. For example,

```
if partyStarted:
    if person in guests:
        print("Welcome")
```

- The above example is equivalent to the following compound conditional:

```
if partyStarted and person in guests:
    print("Welcome")
```

- Python uses a technique known as **short circuiting**, where a partial evaluation of a boolean expression suffices as soon as the outcome can be properly determined.
  x and y: if x is False, the result is False. If x is True, then the result of (x and y) is y.
  x or y: If x is True, the result is True. If x is False, the result of (x or y) is y.
- Here is an example of nested control structures:

```
balance = 0
for entry in transactions:
    balance += entry
    if balance < 0:
        print("Overdraft warning")
```

- **if-else Syntax:** In the simplest form,

```
if condition:
    body1     #Notice the indentation
else:         #No indentation, aligned with if
    body2     #Indentation
```

- Here is the flow-chart:

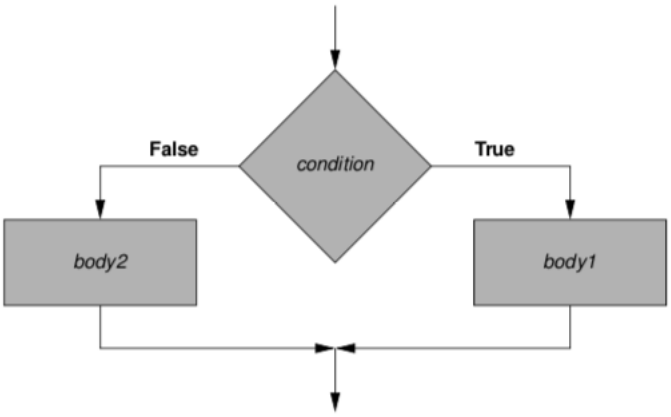


FIGURE 4.13: Flowchart for an **if-else** statement.

- Here is an example:

```
depositTotal = 0
withdrawalTotal = 0
for entry in transactions:
    if entry >0:
        depositTotal += entry
    else:
        withdrawalTotal += entry
```

- **if-elif-else Syntax:** The syntax is as follows:

```
if condition1:
      body1           #Notice the indentation
elif condition2:      #No indentation, aligned with if
      body2           #Indentation
elif condition3:      #No indentation, aligned with if
      body3           #Indentation
.
.
.
else:                 #No indentation, aligned with if
      default-body    #Indentation
```

- Here is a concrete example:

```
daynumber = int(input("Enter an integer between 1 and 7 (inclusive): "))
if daynumber == 1:
    day = "Monday"
elif daynumber == 2:
    day = "Tuesday"
elif daynumber == 3:
    day = "Wednesday"
elif daynumber == 4:
    day = "Thursday"
elif daynumber == 5:
    day = "Friday"
elif daynumber == 6:
    day = "Saturday"
else:
    day = "Sunday"
```

8. Designing and implementing classes – a Fraction class (sections 6.4)

- It is best to learn how to design classes by examples. Here is an example:

```
class EmployeeFirstDraft:
    def info(self):
      print("Name of the employee is unknown")
      print("The annual income of the employee is unknown")
      print("The employee joined our company on unknown.")
```

- Run this module, and then try

```
ted = EmployeeFirstDraft()
ted.info()
holly = EmployeeFirstDraft()
holly.info()
```

- Here is a better class.

```
class Employee:
  def __init__(self, firstName, middleName, lastName,
                                    dateOfJoining, annualIncome):
    self._firstName = firstName
    self._middleName = middleName
    self._lastName = lastName
    self._dateOfJoining = dateOfJoining
    self._annualIncome = annualIncome

  def getName(self):
    name = self._firstName+' '+self._middleName+' '+self._lastName
    return name

  def getDateOfJoining(self):
    return self._dateOfJoining

  def getAnnualIncome(self):
    return self._annualIncome

  def newAnnualIncome(self):
    new = (1.05)*self._annualIncome
    return new

  def info(self):
   print("Name of this employee is", self.getName()+'.')
```

```
    print("Last year his/her annual income was $", str(self.getAnnualIncome())+'.')
    print("This year his/her annual income is $", str(self.newAnnualIncome())+'.')
    print("The employee joined our company on", str(self.getDateOfJoining())+'.')
```

- Now try the following:

```
ted = Employee("Ted", "Matthew", "Bush", "1 January 2019", 45000)
ted.info()
```

- The keyword **class** declares the definition of a new class. The subsequent identifier *Employee* is our choice for naming our class (or EmployeeFirstDraft in the first case). The colon marks the beginning of the block of code that serves as the body of the class definition.
- The first method is named __init__. Informally, this method is referred to as the **constructor.**. Its primary purpose is to establish initial values for the **attributes** of the newly created object. The *implicit* parameter **self** serves internally to identify the parameter instance being constructor.
- Other methods are defined following the same syntax as definitions of functions. Some methods are **accessors** – these access the state information of the instance. Some methods are **mutators**– these change the state information of the instance.
- **A Fraction Class:**. We develop an **immutable** Fraction class. Our internal representation of a fraction consists of two numbers, a numerator and a denominator. We wish to ensure that all fractions are reduced to lowest terms and stored with a nonnegative denominator. A fraction with denominator of zero is viewed arithmetically as an undefined value. We will solely use **special** methods.
- We would like to create instances of this class such as *Fraction(16,9), Fraction(3), Fraction()* to represent $\frac{16}{9}, \frac{3}{1}$ and 0 respectively.

```
# Program: Fraction.py
from gcd import gcd
class Fraction:
  def __init__(self, numerator=0, denominator=1):
    if denominator == 0:          # fraction is undefined
      self._numer = 0
      self._denom = 0
    else:
      factor = gcd( abs(numerator), abs(denominator) )
      if denominator < 0:         # want to divide through by negated
          factor
        factor = -factor
      self._numer = numerator // factor
      self._denom = denominator // factor
  ######## Arithmetic Methods ########
  def __add__(self, other):
```

```
    return Fraction(self._numer * other._denom + self._denom * other.
        _numer,
                    self._denom * other._denom)

  def __sub__(self, other):
    return Fraction(self._numer * other._denom - self._denom * other.
        _numer,
                    self._denom * other._denom)

  def __mul__(self, other):
    return Fraction(self._numer * other._numer, self._denom * other.
        _denom)

  def __truediv__(self, other):
    return Fraction(self._numer * other._denom, self._denom * other.
        _numer)
  ######## Comparison Methods ########
  def __lt__(self, other):
    return self._numer * other._denom < self._denom * other._numer

  def __eq__(self, other):
    return self._numer == other._numer and self._denom == other._denom
  ######## Type Conversion Methods ########
  def __float__(self):
    return float(self._numer) / self._denom

  def __int__(self):
    return int(float(self))    # convert to float, then truncate

  def __str__(self):
    if self._denom == 0:
      return 'Undefined'
    elif self._denom == 1:
      return str(self._numer)
    else:
      return str(self._numer) + '/' + str(self._denom)
```

9. Error checking and exceptions (section 5.5)

- We have encountered various errors in the past. Example, NameError, TypeError, ValueError.
- Each of these types of standard errors is a subclass of a general Exception class.
- Exceptions are used to report scenarios that are out of the ordinary. The result of an exception is to immediately stop the current executing code.
- For instance, if your code involves the following:

```
number = int(input("Enter a number from 1 to 10:  "))
```

- You would like to make certain that the user inputs a number from 1 to 10.

```
number = 0
while not (1 <= number <= 10):
    try:
        number = int(input("Enter a number from 1 to 10:  "))
        if not (1 <= number <= 10):
        print("Your number must be from 1 to 10.")
    except ValueError:
        print("That is not a valid integer.")
```

- This approach works well, but does not know what to do with other kind of errors.

```
number = 0
while not (1 <= number <= 10):
    try:
        number = int(input("Enter a number from 1 to 10:  "))
        if not (1 <= number <= 10):
        print("Your number must be from 1 to 10.")
    except ValueError:
        print("That is not a valid integer.")
    except EOFError:
        print("What are you doing?")
    except:
         print("Looks like you do not know what an integer is.")
         print("That is okay. We will choose.")
         number = 7
```

- **Raising an Exception**
- We can also raise exceptions from within our own code. For example:

```
def sqrtE(number):
    if number <0:
        raise ValueError("sqrt(number): number is negative")
    prevGuess = 0.0
```

```
    guess = min(1.0, number)
    while prevGuess < guess:
        prevGuess = guess
        avg = (guess + number/guess) / 2.0
        guess = min(avg, number/avg)
    return guess
```

- When raising an exception, we technically create an instance of the appropriate error class, providing a descriptive error message as a parameter to its constructor.
- **Type Checking**
- A ValueError happens when a function or method obtains the appropriate type of argument but yet, an inappropriate one. Like in our example, the square-root of a negative number. A TypeError occurs when a function or method obtains an argument of the inappropriate type.
- We want to be able to check that the data sent matches the expected type.
- Our preferred tool for type checking is a built-in function, isinstance. The syntax of isinstance() is

```
isinstance(object, classinfo)
```

- Here, classinfo could be a class, type, or tuple of classes and types.
- For instance, isinstance(number, float) checks whether the number is a floating-point value.
- We now improve our square-root function as follows:

```
def sqrtE(number):
    if not (isinstance(number, (int, float))):
        raise TypeError("sqrt(number): number must be numeric")
    if number <0:
        raise ValueError("sqrt(number): number is negative")
    prevGuess = 0.0
    guess = min(1.0, number)
    while prevGuess < guess:
        prevGuess = guess
        avg = (guess + number/guess) / 2.0
        guess = min(avg, number/avg)
    return guess
```

- Here is another function. What do you think it does?

```
1  def scaleData(data, factor):
2    if not isinstance(data, list):
3      raise TypeError('data must be a list')
4    for item in data:
5      if not isinstance(item, (int,float)):
6        raise TypeError('data items must all be numeric')
7    if not isinstance(factor, (int,float)):
8      raise TypeError('factor must be numeric')
9    for i in range(len(data)):
10     data[i] = factor * data[i]
```

10. Design, Documentation, Unit Testing (sections 7.2, 7.4, 7.5, 7.6, 7.7)

- **Top-Down Design and Bottom-Up Implementation:**
- Before we write any portion of a large program we should consider its overall design.
- We need to identify potential classes and the way in which objects from those classes will interact.
- A starting point is to envision the use of our final product and design a top-level class and its interface.
- From there we expand our design to include classes that the top-level class will need in order to perform its tasks.
- We repeat this process until we have designed a collection of individual components that can be written and tested independently.
- This approach is known as **top-down design.**
- Once the design is fixed, implementation should start at the lowest level and work up to the top level.
- This technique is known as **bottom-up implementation.**
- Every component must be tested thoroughly before moving on to the next higher level.

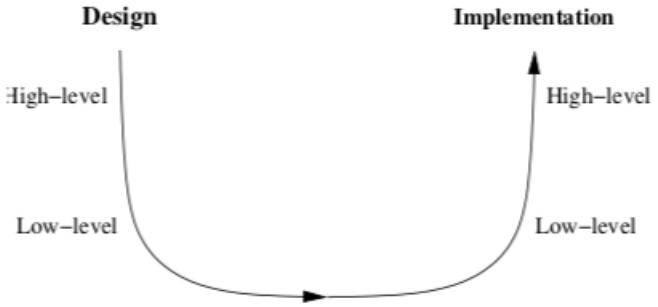


FIGURE 7.2: A development cycle with a top-down design followed by a bottom-up implementation.

- The existence of smaller independent pieces is called **modularity**. One advantage of this design is that the pieces can be implemented and tested in parallel by a team. Furthermore, many of the components may have more general value, allowing for **code reuse** across multiple portions of the program or in other projects.
- **Naming Conventions:**
- Recall, an identifier cannot begin with a numeral, cannot have any spaces, and cannot be among a short ist of reserved keywords. Identifiers are case sensitive.
- A name should be descriptive, clearly suggesting an underlying purpose so as to improve the overall readability of the code. A thoughtful choice of a name can go along way in establishing correct intuition.
- **Classes:** A class name should be a singular noun, and begin with a capital letter. When the class name involves two or more English words, we concatenate those words while capitalizing the first letter of each word (e.g. CannonBall).
- **Functions:** The name of a function, whether stand-alone or a method of a class, should be a verb and begin with a lowercase letter, but each additional word is capitalized (eg. jumPrevChannel, getYCoord).

- **Data:** A data name should be a noun, begin with a lowercase letter, but each additional word is capitalized. Data name may be singular or plural depending on its usage. Choose names that clearly indicate what type of data is being stored and how it will be used (eg. guestList, idNumber).
- **Parameters:** Names of parameters follow the same convention as for data.
- **The advantage of naming conventions:** Following conventions helps us to tell exactly what a name represents even with limited context.
- **Formal Documentation: Python Docstrings**
- Documentation informs another programmer how to properly use your classes and functions, and serves as a formal specification of promised behavior that must be internally implemented. The expected outward behavior of a class or a function should be well defined and documented before the actual implementation is written.
- The symbol # is used for directly embedding comments within source code. This style of comment is ignored by the interpreter yet visible to a programmer, making it very valuable for someone who reads a piece of code.
- To access documentation without going into the code, Python supports another style of documentation using well-placed strings known as **docstrings**.
- These strings are visible within the source code and they are also used to generate documentation seen in the Python interpreter through the use of the *help* command and to generate documentation on webpages using a corresponding utility *pydoc*.
- A docstring is technically a string literal that occurs as the very first element within the body of a class or function.
- Triple quote delimiters (" " ") are used since these allow for multiline strings.
- A docstring should begin with a brief one line description. If further explanation is warranted, such as the purpose or type of parameters and return value, that information should be provided after a subsequent blank line within the docstring.
- **Encapsulation of Implementation Details**
- The principle of treating internal implementation details of a software component as private is known as **encapsulation.**
- The primary advantage of encapsulation is that it limits the interdependence between software components.
- Having a clear designation of the public and private aspects of the component benefits both the author and the client using the program. For the author, the encapsulation of private details provides greater flexibility in the development and maintenance of the software. For the client, the identification of the public portion draws focus to only that which must be understood to properly use the component.
- **High-Level Design:** The use of encapsulation begins with the creation of the high-level design for a software project. ıThe design of the various public interfaces depends upon the degree of information sharing that is necessary.
- **Underscored Names in Python:** Python supports the notion of encapsulation through naming conventions.

- If the identifier given to a class, or to a method or attribute of a class, begins with an alphabetic character that element is presumed to be of public interest.
- To designate something as private, we choose an identifier that begins with a single underscore character.
- When methods of a class are named with a leading underscore, they are not displayed in documentation generated by the *help* command or the *pydoc* utility.
- When the wildcard form of an import is performed (eg. from cs1graphics import *), only the public elements of the module are loaded.
- But note, if a user is aware of an underscored name, then they can be directly accessed (eg. point._xcoord = 5).
- Note that the single underscore naming convention is unrelated to the double underscore naming used for special methods such as __init__. Also, local variables within a function body are not typically underscored, as they are already private due to their local scope.
- **Data Members:** In order to protect the integrity of an object's state, we generally encapsulate all attributes of a class as private, preferring public access through designated accessors and mutators.
- **Private Methods:** We can designate certain methods as private by using underscore as the first character of their names. While the public methods are ones that we expect to be called by others, private methods are used for our own convenience when implementing a class; they should only be called from within the remainder of the class. Often these functions are used to perform some common task for the rest of the class to improve the overall organization.
- **Modules and Unit Testing:**
- The source code for a large project is typically split into several files. This helps improve the organization, supports reuse of code, and allows for different developers to edit different components with less interference.
- If one class relies upon definitions from a separate file, the latter elements must be imported into the former before they can be used.
- Another important principle of good software development is to test each individual component as it is written. Testing a class individually, instead of as part of the larger program, is called **unit testing**. This is the essence of bottom-up implementation.
- With unit testing earlier classes are rendered reasonably error free before being used.
- Python provides a convenient manner for embedding unit tests inside the same file as the class implementation. The basic format is:

```
class ClassName:
    #implementation here

if __name__ = '__main__':
  #unit test here
```

- The conditional allows us to execute commands as part of an isolated unit test, yet to have those commands ignored when the file is being incorporated into the larger software project.

- When a Python program is executed, the flow of control begins with an indicated module. That top-level module may import other modules along the way.
- Within an individual module, the special variable __name__ serves to identify the context in which that module is currently being loaded. If the module is the one upon which the interpreter was initially invoked, this variable will be automatically set to the string '__main__'. Alternatively, if the module is being imported from elsewhere, then __name__ will be set to this module's own name.

- **Standard Input and Output**
- The simplest form for receiving input from a user is through the use of **input** function:

```
identifier = input("Question: ")
#Or,
identifier = input() #Here, no question is displayed.
```

- The system waits for the user to type. Once the keys are typed, the user presses the enter key. The input function returns the string of entered characters, up to but not including the final newline. Here are a few examples:

```
color = input("What is your favorite color?")


#Or
color = input("What is your favorite color? [blue] ")
if color == "":
    color = "blue"


 #Or
color = input("What is your favorite color? [blue] ")
if  not color:   #empty string has boolean value of False
    color = "blue"
```

- The simplest form for generating output is by using the print command. Here are some simple examples:

```
print("I like", favoriteFruit, "and the number", myAge)
```

- By default setting an explicit space is automatically inserted into the output between successive arguments separated by commas. The print command also generates one final newline character ('\n')
- **Formatted Strings**
- Here are various ways of using **string formatting**. The following commands will all result in the same output.

```
name = "Myteam"
rank = 1
total = 20
print(name+": ranked", rank, "of",total, "teams.")
print("%s: ranked %d of %d teams."%(name, rank, total))
print(f"{name}: ranked {rank} of {total} teams.")
print("{0}: ranked {1} of {2} teams.".format(name, rank, total))
```

- In the above formatting, % d represents integer (even though the term is decimal); % s represents string or list, tuple, % i can also be used for integer, %f for floating-point, %% to print % symbol.
- Note % 4d causes an integer to be printed with a minimum of four characters. If the integer is less than 4 digits long, then it is right justified by default. Check what happens with the following (some are in fact syntactically incorrect – which ones?):

```
print("Give me $%10d .")%3456)
print("Give me $%-10d .")%3456)
print("Give me $%010d .")%3456)
print("Give me $%10d .")%3456789876543212)
print("Give me $%10f .")%3456)
print("Give me $%10.2f .")%3456)
print("Give me $%10d .")%3456/1312)
print("Give me $%10f."%3456/1312)
print("Give me $%10.2f .")%3456/1312)
print("Hello %s."%'Amy')
print("Hello %10s."%'Amy')
print("Hello %-10s."%'Amy')
print("Give me ${0}.".format(3456))
print("Give me ${0:10}.".format(3456))
print("Give me ${0:>10}.".format(3456))
print("Give me ${0:<10}.".format(3456))
print("Give me ${0:^10}.".format(3456))
print("Give me ${0:10.2}.".format(3456))
print("Give me ${0:10.2}.".format(3456.45678))
print("Give me ${0:10.2f}.".format(3456.45678))
print("Give me ${0:0.2f}.".format(3456.45678))
print("Give me ${0:10.20f}.".format(3456.45678))
```

- **Working with files**
- Python supports a built-in class named *open* to manipulate files on the computer. The constructor for Python's *open* requires a string parameter, identifying the underlying filename on the computer system.

```
myfile = open("filename.extension")
```

- By default, a newly instantiated file object provides *read-only* access to an existing file.
- If no file exists with the specified name or if such a file is unreadable, the call to the constructor results in an IOError.

- Here are several modes of opening files.

```
myfile = open("filename.extension", "r")  #Read mode
myfile = open("filename.extension", "w")  #(Over)Write mode
myfile = open("filename.extension", "a")  #Append mode
```

- Here are some methods of Python's file class. Construct a file, and play with **every command** you see in this table.

Syntax	Semantics
close( )	Disconnects the file object from the associated file (saving the underlying file if necessary)
flush( )	Flushes buffer of written characters, saving to the underlying file.
read( )	Returns a string representing the (remaining) contents of the file.
read(size)	Returns a string representing the specified number of bytes next in the file.
readline( )	Returns a string representing the next line of the file.
readlines( )	Returns a list of strings representing the remaining lines of the file.
write(s)	Writes the given string to the file. No newline is added.
writelines(seq)	Writes each of the strings to the file. No newlines are added.
for line **in** f	Iterates through the file~f, one line at a time.

FIGURE 8.2: Selected behaviors of Python's file class.

- Note, the *flush()* method **saves an open file**, while the *close()* method **saves and closes the file**.
- The *write()* method is slightly different from the print(string, file = filename) approach. While *print()* command automatically converts nonstring arguments to strings, the *write()* method does not. Also note, *write()* does not end in a newline.

```
filename.write(3.14) #this will not work
filename.write(str(3.14)) #this will work
filename.write("%5.2f"%3.14) #this will work
```

- Here are some codes you should try.

```
#Here is the first code
filename = input("What is the filename? ")
source = open(filename)
text = source.read()
source.close()
numchars = len(text)
numwords = len(text.split())
numlines = len(text.split('\n'))
print("Number of characters =", numchars)
print("Number of words =", numwords)
print("Number of lines =", numlines)
```

```
#Here is the second code
filename = input("What is the filename? ")
source = open(filename)

numchars = numwords = numlines = 0
line = source.readline() #Get the first line.
while line:  #Note, line has a boolean value
    numchars += len(line)
    numwords += len(line.split())
    numlines += 1

    line = source.readline() #Get the next line.

filename.close()
print("Number of characters =", numchars)
print("Number of words =", numwords)
print("Number of lines =", numlines)

#Here is the third code
filename = input("What is the filename? ")
source = open(filename)

numchars = numwords = numlines = 0
for line in source:
    numchars += len(line)
    numwords += len(line.split())
    numlines += 1

print("Number of characters =", numchars)
print("Number of words =", numwords)
print("Number of lines =", numlines)
```

- The *input()* command receives input from the user, but the string does not end in '\n' character. On the other hand, the *readline()* method returns the next line of the file including any explicit new line character. Now try what happens with the following:

```
animal = 'dog\n'
animal.rstrip('\n')
animal
fruit = 'apple'
fruit.rstrip('e')
flower = 'rose'
```

```
flower.rstrip('se')
```

- **Case Studies** Here are some robust functions to open, write, or read files. Understand every line of the code and implement them.

```
# Program: FileUtilities.py
#
#
"""A few utility functions for opening files."""
def openFileReadRobust():
    """Repeatedly prompt user for filename until successfully
       opening with read access.

    Return the newly open file object.
    """
    source = None
    while not source:                    # still no successfully
       opened file
      filename = input('What is the filename? ')
      try:
        source = open(filename)
      except IOError:
        print('Sorry. Unable to open file', filename)
    return source

def openFileWriteRobust(defaultName):
    """Repeatedly prompt user for filename until successfully
       opening with write access.

    Return a newly open file object with write access.

    defaultName      a suggested filename. This will be offered
       within the prompt and
                     used when the return key is pressed without
                        specifying another name.
    """
    writable = None
    while not writable:                  # still no successfully
       opened file
      prompt = 'What should the output be named [%s]? '% defaultName
      filename = input(prompt)
```

```
    if not filename:                    # user gave blank response
      filename = defaultName            # try the suggested default
    try:
      writable = open(filename, 'w')
    except IOError:
      print('Sorry. Unable to write to file', filename)
  return writable

def readWordFile():
  """Open and read a file of words.

  Return the list of strings.
  The file format is expected to be with one word specified per
     line.
  """
  print("About to read list of words from file.")
  wordfile = openFileReadRobust()

  words = list()
  for entry in wordfile:
    words.append(entry.rstrip('\n'))

  return words
```

- **Annotating a File**
- We would like to annotate files. That is, a file which looks like

```
Dear Editor,

  I greatly enjoyed your article on Python.

Sincerely,
David
```

- should appear as

```
1  Dear Editor,
2
3    I greatly enjoyed your article on Python.
4
5  Sincerely,
6  David
```

- Here is a program:

```
from FileUtilities import openFileReadRobust, openFileWriteRobust

print("This program annotates a file, by adding")
print("Line numbers to the left of each line. \n")

source = openFileReadRobust()
annotated = openFileWriteRobust('annotated.txt')

#process the file
linenum = 1
for line in source:
    annotated.write('%4d %s'%(linenum, line))
    linenum += 1
source.close()
annotated.close()
print("The annotation is complete.")
```

- In this section we work with the wrapper *cs1graphics.py* designed by the authors of the book. This package is for developing GUI codes. Refer to the appendix for codes using TKinter directly.
- First, download *cs1graphics.py* and move it to the folder you are going to work in for programs involving graphics.
- **The Canvas**
- We can create a canvas by calling the constructor as follows:

```
Canvas()
```

- By itself, a canvas cannot be interacted with. So, it is better to start with an assignment.

```
paper = Canvas()
```

- By default, a newly created canvas is 200 pixels wide and 200 pixels tall, has a white background, and is titled "Graphics Canvas." But canvas is mutable:

```
paper.setBackgroundColor('skyBlue')
paper.setWidth(300)
paper.setTitle('My World')
```

- All of this can be achieved by using **optional parameters**:

```
paper = Canvas(300, 200, 'blue', 'My World')
```

- Here is an overview of the Canvas class:

Canvas	
Canvas(w, h, bgColor, title, autoRefresh)	add(drawable)
getWidth( )	remove(drawable)
setWidth(w)	clear( )
getHeight( )	open( )
setHeight(h)	close( )
getBackgroundColor( )	saveToFile(filename)
setBackgroundColor(color)	setAutoRefresh(trueOrFalse)
getTitle( )	refresh( )
setTitle(title)	wait( )

FIGURE 3.2: Overview of the Canvas class.

- Note, the coordinate system has origin at the top left corner:

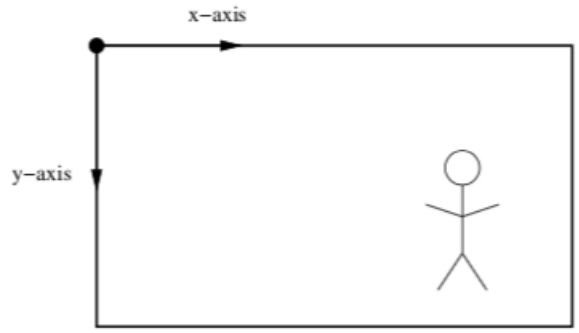


FIGURE 3.3: The canvas coordinate system, with the origin at the top left corner.

- Every graphical object we place upon a canvas is done so using the coordinates of a key **reference point**. For example, the center of a circle.
- Here is a way to create a circle object and add it to our canvas.

```
sun = Circle(30, Point(250,50))
paper.add(sun)
sun.setFillColor('yellow')
```

- Note, the *Circle()* by default is centered at $(0, 0)$ and radius 10 pixels. So, we could have done the following:

```
sun = Circle()
paper.add(sun)
sun.setRadius(30)
sun.move(250,50)
sun.setFillColor('yellow')
```

- **Square**
- A square is another example of a FillableShape. By default, a square is 10 x 10 and centered at $(0, 0)$. Other sizes and central points have to be specified as follows:

```
facade = Square(60, Point(140,130))
facade.setFillColor('white')
paper.add(facade)
```

- We may also resize the size of a square by *facade.setSize(90)*.
- **Rectangle**
- By default, a rectangle has width of 20 pixels, a height of 10 pixels, and is centered at the origin. However, we can specify the width, the height, and the center point, in that order.

```
chimney = Rectangle(15, 28, Point(155,85))
chimney.setFillColor('red')
paper.add(chimney)
```

- By default, our fillable objects have black outlines, one pixel wide. One way to make the boundary disappear is to use the method *setBorderWidth(0)*.

- **Polygon**
- The polygon is a fillable object which is built with a sequence of points written in order (clockwise or counterclockwise) . By default, the polygon's reference point is its first point.

```
tree = Polygon(Point(50,80), Point(30,140), Point(70,140))
tree.setFillColor('darkGreen')
paper.add(tree)
```

- A polygon can also be built by adding one point at a time.

```
tree = Polygon()
tree.addPoint(Point(50,80))
tree.addPoint(Point(30,140))
tree.addPoint(Point(70,140))
tree.setFillColor('darkGreen')
paper.add(tree)
```

- Note, the points above have been added in a sequence with index 0, 1, 2, ... Thus, we may insert a point (*tree.setPoint(Point(50,70),0)*) or change a point (*tree.setPoint(Point(50,70),0)*) or delete a point at a certain index (*tree.deletePoint(i)*).
- **Path**
- A Path connects points just like a polygon, except, a path does not connect the last point point to the first, and a path is not fillable. A path supports **setBorderColor** and **setBorderWidth** methods. For instance,

```
sunraySW = Path(Point(225,75), Point(210,90))
sunraySW.setBorderColor('yellow')
sunraySW.setBorderWidth(6)
paper.add(sunraySW)
```

- **Colors**
- A color is represented behind the scene by what is known as its **RGB value.** This is a tuple of three numbers that represent the intensity of red, green, and blue respectively, using a scale from 0 (no intensity) to 255 (full intensity). For example, 'skyBlue' is RGB of (136,206,235).
- **Depths**
- The relative ordering of conflicting shapes is controlled by an underlying numeric attribute representing the depth of each drawable object. By default, all objects are assigned a depth value of 50 and their relative ordering is is arbitrary. However, their depths can be changed to control the image. The object with a smaller depth is drawn above the object with the greater depth. So, we draw grass with the greatest depth.

```
grass = Rectangle(300,80, Point(150,160))
grass.setFillColor('green')
grass.setBorderColor('green')
grass.setDepth(75)
```

```
paper.add(grass)
```

```
1  from cs1graphics import *
2  paper = Canvas(300, 200, 'skyBlue', 'My World')
3
4  sun = Circle(30, Point(250,50))
5  sun.setFillColor('yellow')
6  paper.add(sun)
7
8  facade = Square(60, Point(140,130))
9  facade.setFillColor('white')
10 paper.add(facade)
11
12 chimney = Rectangle(15, 28, Point(155,85))
13 chimney.setFillColor('red')
14 chimney.setBorderColor('red')
15 paper.add(chimney)
16
17 tree = Polygon(Point(50,80), Point(30,140), Point(70,140))
18 tree.setFillColor('darkGreen')
19 paper.add(tree)
20
21 smoke = Path(Point(155,70), Point(150,65), Point(160,55), Point(155,50))
22 paper.add(smoke)
23
24 sunraySW = Path(Point(225,75), Point(210,90))
25 sunraySW.setBorderColor('yellow')
26 sunraySW.setBorderWidth(6)
27 paper.add(sunraySW)
28 sunraySE = Path(Point(275,75), Point(290,90))
29 sunraySE.setBorderColor('yellow')
30 sunraySE.setBorderWidth(6)
31 paper.add(sunraySE)
32 sunrayNE = Path(Point(275,25), Point(290,10))
33 sunrayNE.setBorderColor('yellow')
34 sunrayNE.setBorderWidth(6)
35 paper.add(sunrayNE)
36 sunrayNW = Path(Point(225,25), Point(210,10))
37 sunrayNW.setBorderColor('yellow')
38 sunrayNW.setBorderWidth(6)
39 paper.add(sunrayNW)
```

FIGURE 3.10: Complete source code for drawing our house (continued on next page).

```
40 grass = Rectangle(300, 80, Point(150,160))
41 grass.setFillColor('green')
42 grass.setBorderColor('green')
43 grass.setDepth(75)                # must be behind house and tree
44 paper.add(grass)
45
46 window = Rectangle(15, 20, Point(130,120))
47 paper.add(window)
48 window.setFillColor('black')
49 window.setBorderColor('red')
50 window.setBorderWidth(2)
51 window.setDepth(30)
52
53 roof = Polygon(Point(105, 105), Point(175, 105), Point(170,85), Point(110,85))
54 roof.setFillColor('darkgray')
55 roof.setDepth(30)                # in front of facade
56 chimney.setDepth(20)             # in front of roof
57 paper.add(roof)
```

FIGURE 3.10 (continuation): Complete source code for drawing our house.

- **Text and Image classes**
- The Text class is used for rendering character strings within the drawing area of the canvas. The syntax is

```
Text(string, font size =12)
```

- Note, the way to write a text to a canvas object is:

```
announcement = Text('This is my announcement')
announcement.move(100,180)
canvasExample.add(announcement)
```

- The reference point for a text object is the center of the text.
- The library cs1graphics also includes an Image class that provides support for using a raw image loaded from a file. An image object is constructed

```
myImage = Image('lightbulb.gif') #Can also be jpg, bmp, tiff)
myImage.move(100,180)
canvasExample.add(myImage)
```

- The reference point for an image object is aligned with the top left corner of the image.
- **Rotating, Scaling, and Flipping**
- All Drawable objects can be rotated, scaled, or slipped. The reference point of a shape stays fixed during the transformation.
- Note, there is a method to move the reference point without moving the object. The method is *adjustReference(dx,dy)*. Note, we may even move the reference point away from the shape itself. This adjusting of the reference point is useful for the ensuing rotation.

- **Rotating:** The *rotate* method rotates an object clockwise around the reference point with the specified angle given in degrees. For example,

```
diamond = Square(40, Point(100,100))
diamond.rotate(20)

#Or, change the reference point first
block = Square(40, Point(100,100))
block.adjustReference(-20,20)
block.rotate(20)
```

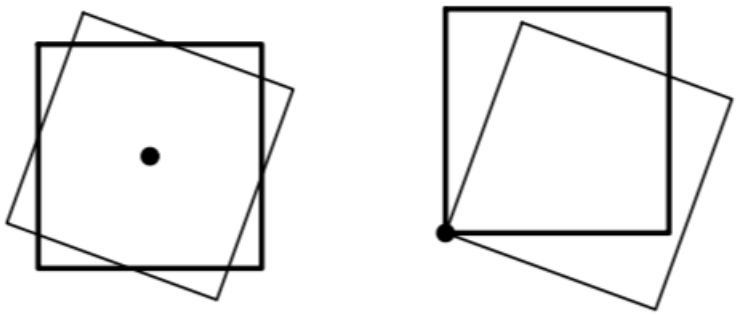


FIGURE 3.11: Rotating a square 20° in the clockwise direction. In both diagrams the original position is drawn with darker border and the resulting position with lighter border. In the case on the left, the square is rotated about its center point. On the right, the same square is rotated about its lower left corner.

- **Scaling:** All Drawable objects support a method *scale* that takes a single parameter specifying a positive multiplicative factor by which the object is to be scaled. Note, if the parameter is greater than 1, then the object enlarges while if the parameter is less than 1, then the object shrinks. The reference point remains fixed.

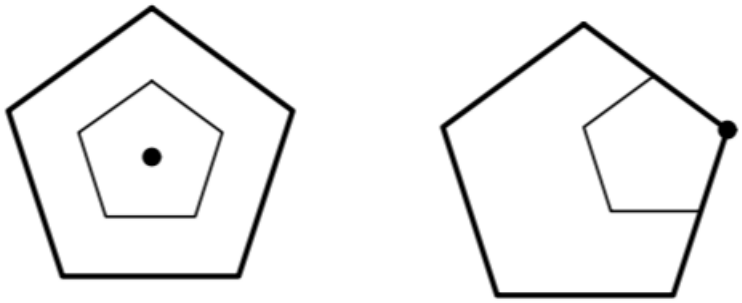


FIGURE 3.12: Scaling a pentagon about two different reference points. In both diagrams the original position is drawn with darker border and the resulting position with lighter border. On the left, the pentagon is scaled relative to its center point. On the right, the pentagon is scaled about the rightmost corner.

- **Flipping:** All Drawable objects support a method *flip()* which reflects the object about the vertical line through the reference point. The method *flip()* also accepts an angle in degrees as a parameter this rotation the line of reflection.

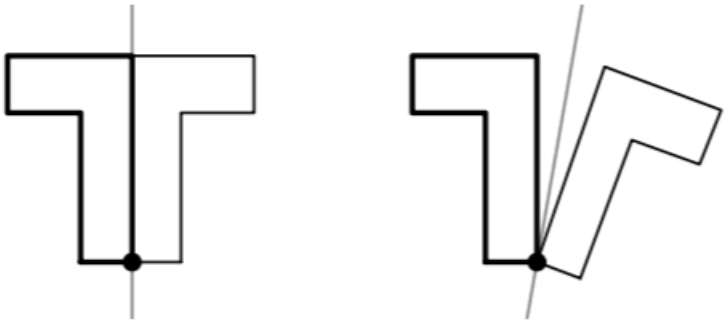


FIGURE 3.13: Flipping a flag about two different axes of symmetry. In both diagrams the original position is drawn with darker border and the resulting position with lighter border. On the left, the flag is flipped about the vertical axes by default with flip( ). The right shows the result of flip(10).

- **Cloning:** The Drawable types support a *clone()* method that returns a brand new copy. The clone has precisely the same settings as the original object, is not automatically added to any canvas. Once created, the clone can be manipulated independent of the original. For example,

```
otherTree = tree.clone()
otherTree.move(170,30)
otherTree.scale(1.2)
paper.add(otherTree)
```

- The next page gives an overview of the Drawable objects.

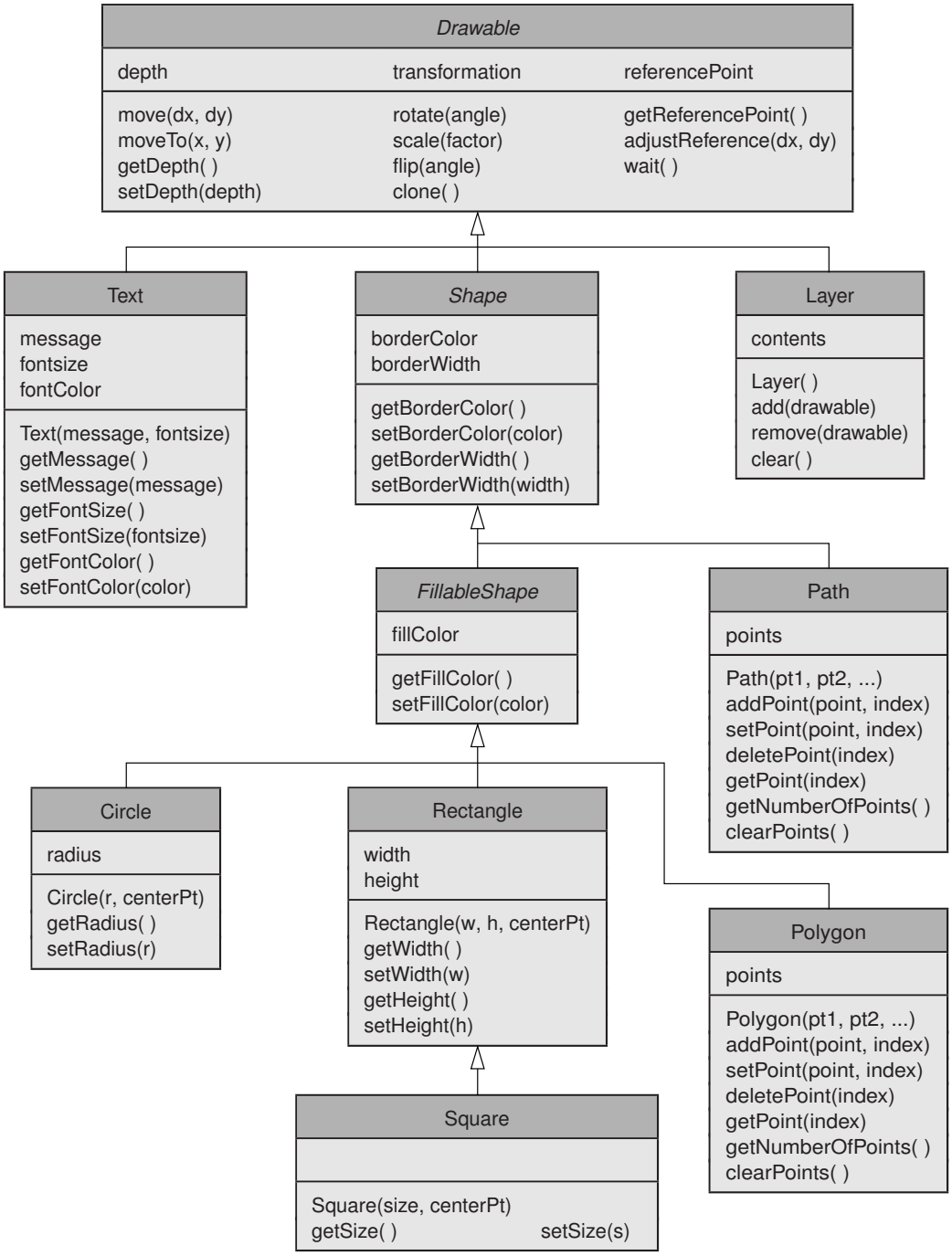


FIGURE 3.4: An overview of the Drawable objects.

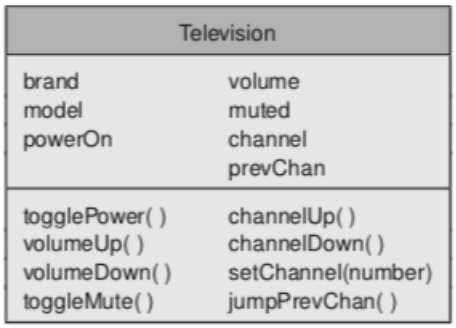
- We introduce the concept of **inheritance**. This is a technique that allows us to define a new (child) class based upon an existing (parent) class. The child class inherits all of the members of its parent class, thereby reducing duplication of existing code.
- We typically differentiate a child class from its parent in two was.
- The child may introduce one or more behaviors beyond those that are inherited, thereby **augmenting** the parent class.
- A child class may also **specialize** one or more of the inherited behaviors from the parent. This specialization is accomplished by providing an alternative definition for the inherited method, thereby **overriding** the original definition.
- Note, a child class might specialize certain existing behaviors while introducing others.
- Let us start with an example. First, let us build the Television class.

```
# Program: Television.py
# Authors: Michael H. Goldwasser
#          David Letscher
#
class Television:
  def __init__(self,brand,model):
    self._brand = brand
    self._model = model
    self._powerOn = False
    self._muted = False
    self._volume = 5
    self._channel = 2
    self._prevChan = 2

  def togglePower(self):
    self._powerOn = not self._powerOn

  def toggleMute(self):
    if self._powerOn:
      self._muted = not self._muted

  def volumeUp(self):
    if self._powerOn:
      if self._volume < 10:
        self._volume += 1
      self._muted = False
      return self._volume
```

```
def volumeDown(self):
  if self._powerOn:
    if self._volume > 0:
      self._volume -= 1
    self._muted = False
    return self._volume

def channelUp(self):
  if self._powerOn:
    self._prevChan = self._channel     # record the current value
    if self._channel == 99:
      self._channel = 2                # wrap around to minimum
    else:
      self._channel += 1
    return self._channel

def channelDown(self):
  if self._powerOn:
    self._prevChan = self._channel     # record the current value
    if self._channel == 2:
      self._channel = 99               # wrap around to maximum
    else:
      self._channel -= 1
    return self._channel

def setChannel(self, number):
  if self._powerOn:
    if 2 <= number <= 99:
      self._prevChan = self._channel  # record the current value
      self._channel = number
    return self._channel

def jumpPrevChannel(self):
  if self._powerOn:
    incoming = self._channel
    self._channel = self._prevChan
    self._prevChan = incoming
    return self._channel

def __str__(self):
```

```
        display = 'Power setting is currently ' + str(self._powerOn) +
            '\n'
        display += 'Channel setting is currently ' + str(self._channel
            ) +'\n'
        display += 'Volume setting is currently ' + str(self._volume)
            +'\n'
        display += 'Mute is currently ' + str(self._muted)
        return display
```

- Here is a Television class dagram:



- **Augmentation:** We wish to design a deluxe version of our television. We wish to add support for managing a set of so called favorite channels. In particular, we have three new behaviors in mind:

  addToFavorites() - a method that adds the currently viewed channel to the set of favorites (if not already there);

  removeFromFavorites() - a method that removes the currently viewed channel from the set of favorites (if it is present).

  jumpToFavorite() - a method that jumps from the current channel setting to the next higher channel in the set of favorites. However, if no favorites are numbered higher than the current channel, it wraps around to the lowest favorite. If there are no favorites at all, the channel remains unchanged.

- For this we need to import Television class from Television.py. We develop a new class *DeluxeTV* that uses *Television* class as its parent. The syntax is as follows:

```
class DeluxeTV(Television):   #Inheritance is indicated by placing
    the parent class in parentheses
```

- With such a declaration, the new child inherits all existing methods of the parent class. We need to specify the augmented functionality. So, the __init__ method is as follows:

```
def __init__(self):
    Television.__init__(self)
    self._favorites = []
```

- The state of a deluxe television includes all the attributes from Television, and now an additional list of favorites.

- We then need to add the three functions as specified initially. Here is the entire definition for the DeluxeTV class.

```
# Program: DeluxeTV1.py
# Authors: Michael H. Goldwasser
#          David Letscher
#
#
from Television import Television

class DeluxeTV(Television):
  """A television that maintains a set of favorite channels."""

  def __init__(self,brand,model):
    """Creates a new DeluxeTV instance.

    The power is initially off, yet when turned on the TV
    is tuned to channel 2 with a volume level of 5. The set
    of favorite channels is initially empty.
    """
    Television.__init__(self,brand,model)        # parent
       constructor
    self._favorites = []

  def addToFavorites(self):
    """Adds the current channel to the list of favorites,
    if not already there.

    If power is off, there is no effect.
    """
    if self._powerOn and self._channel not in self._favorites:
      self._favorites.append(self._channel)

  def removeFromFavorites(self):
    """Removes the current channel from the list of favorites,
    if present.

    If power is off, there is no effect.
    """
    if self._powerOn and self._channel in self._favorites:
      self._favorites.remove(self._channel)
```

```
def jumpToFavorite(self):
    """Jumps to the "next" favorite channel as per the following
      rules.

    In general, this method jumps from the current channel setting
    to the next higher channel which is found in the set of
    favorites. However if no favorites are numbered higher than
    the current channel, it wraps around to the lowest favorite.
    If there are no favorites, the channel remains unchanged.

    Returns the resulting channel setting.

    If power is off, there is no effect.
    """
    if self._powerOn and len(self._favorites)>0:
      closest = max(self._favorites)     # a guess
      if closest <= self._channel:       # no bigger channel exist
        closest = min(self._favorites)   # wrap around to min
      else:                              # let's try to get closer
        for option in self._favorites:
          if  self._channel < option < closest:
            closest = option             # a better choice
      self.setChannel(closest)           # rely on inherited method
      return closest
```

- **Specialization**
- One reason that the *jumpToFavourite()* method is complicated is because the collection of favorite channels is unsorted within the internal **list**. This motivates the development of a new class SortedSet.
- Our SortedSet will maintain a set of elements while ensuring that duplicates are removed and that the elements are ordered. While the list class is not ideal, it is still useful. So, we start with

```
class SortedSet(list):
```

- Note, we do not need any additional attributes. But we might want to override *insert(index, object)* since it might interfere in the sorting process or introduce duplication. So we have to define our own *insert* method.
- We also introduce a new method *indexAfter(value)* to determine the proper index for inserting a new element into a set. This method returns the index of the first element that is strictly larger than the given parameter. When there is no such value it returns the length of the entire set.

```
def indexAfter(self, value):
    walk = 0
    while walk < len(self) and value >= self[walk]:
        walk += 1
    return walk
```

- Notice that the *indexAfter* method is a public method because it seems useful enough to be used in other programs. Also note, since our class is inherited from **list** class, our instance is a list. So the syntaxes len(self) and self[index] are already defined.
- Now, we define *insert* method:

```
def insert(self, value):
    if value not in self:
        place = self.indexAfter(value)
        list.insert(self, place, value)
```

- Notice the last line explicitly invokes the parent **list.insert**.
- We also need to override the *append* method of the **list** class.

```
def append(self, object):
    self.insert(object)  #this is the insert from SortedSet
```

- The *insert* method above is from SortedSet since the object is specifically from the SortedSet class. The parent's methods are only applied in cases where there is no duly named method explicitly in the native context.
- This is an example of the principle of **polymorphism** and a key to the object-oriented paradigm. When the caller invokes a method on an object, the actual behavior depends upon the precise type o f the given object.
- Here is a complete SortedSet class which we will need to build a better DeluxeTV class.

```
# Program: SortedSet.py
# Authors: Michael H. Goldwasser
#          David Letscher
class SortedSet(list):
  """
  Maintains an ordered set of objects (without duplicates).
  """

  def __init__(self, initial=None):
    """Default constructor creates an empty SortedSet.

    If initial sequence is given as parameter, creates
    initial configuration using those elements, with
    duplicates removed.
    """
```

```
    list.__init__(self)   # calls the parent constructor
    if initial:
      self.extend(initial)

  def indexAfter(self, value):
    """
    Find first index of an element strictly larger than given
    value.

    If no element is greater than given value, this returns the
    length of the set.
    """
    walk = 0
    while walk < len(self) and value >= self[walk]:
      walk += 1
    return walk

  def insert(self, value):
    """
    Adds given element to the sorted set.

    If the value is already in the set, this has no effect.
    Otherwise, it is added in the proper location.

    value   element to be added to the set.
    """
    if value not in self:            # avoid duplicates
      place = self.indexAfter(value)
      list.insert(self, place, value) # the parent's method

  def append(self, object):
    """
    Identical to insert(object).
    """
    self.insert(object)
```

- Here is a second version of DeluxeTV class:

```
# Program: DeluxeTV2.py
# Authors: Michael H. Goldwasser
#          David Letscher
##
from Television import Television
from SortedSet import SortedSet

class DeluxeTV(Television):
  """
  Just like a standard television, but with additional
  support for maintaining and using a set of favorite channels.
  """

  def __init__(self,brand,model):
    """
    Creates a new Television instance.

    The power is initially off.  Upon the first time the TV is
    turned on, it will be set to channel 2, and a volume level
    of 5.

    The television maintains a list of favorite channels,
    initially empty.
    """
    Television.__init__(self,brand,model)          # parent
       constructor
    self._favorites = SortedSet()

  def addToFavorites(self):
    """
    Adds the current channel to the list of favorites, if not
    already there.

    If power is off, there is no effect.
    """
    if self._powerOn:
      self._favorites.append(self._channel)

  def removeFromFavorites(self):
```

```
    """
    Removes the current channel from the list of favorites,
    if present.

    If power is off, there is no effect.
    """
    if self._powerOn and self._channel in self._favorites:
      self._favorites.remove(self._channel)

  def jumpToFavorite(self):
    """
    Jumps to the 'next' favorite channel as per the following
    rules.

    In general, this behavior jumps from the current channel
    setting to the next higher channel which is found in the set
    of favorites.  However if there are no such higher channels,
    it jumps to the overall minimal favorite.  If there are no
    favorites at all, this has no effect on the channel setting.

    Returns the resulting channel setting.

    If power is off, there is no effect.
    """
    if self._powerOn and len(self._favorites)>0:
      resultIndex = self._favorites.indexAfter(self._channel)
      if resultIndex == len(self._favorites):
        result = self._favorites[0]        # wrap around
      else:
        result = self._favorites[resultIndex]
      self.setChannel(result)
      return result
```

- **Further details**
- There are some other behaviors from the **list** class which could be overridden. For example:

```
def extend(self, other):
    for element in other:
        self.insert(element)
```

- Recall that the *extend* method of the **list** class modifies the original list by adding all elements of the other list to the end. Now the *extend* method of the **SortedSet** filters out duplicates and maintains the sorted order.
- Recall, when we created the **SortedSet** class , we inherited from the existing **list** class by " class SortedSet(list):" In other words, we relied on the underlying storage mechanism provided by the **list** class. The default constructor provided an empty list which by nature is sorted and without duplicates.
- Now suppose we were to start with an initial list generated from a string; example, *list('hello')*. This list does not result in a sorted set. So, we override __init__ as follows:

```
def __init__(self, initial=None):
    list.__init__(self)   #calls the parent constructor
    if initial:
        self.extend(initial) #this results in a SortedSet list
```

- Next, note that the __add__ method of the **list** class is not a mutator. This method creates and returns a new list that is a composition of two existing lists. This method also needs to be overridden so as to create SortedSet.

```
def __add__(self, other):
    result = SortedSet(self)  #creates a new copy of self
    result.extend(other)      #add other elements of this copy
    return result
```

- The **list** class has its own *sort* method which is time-consuming and unnecessary for our **SortedSet** class. So we do the following:

```
def sort(self):
    pass.  #do nothing
```

- The **list** class has two other methods *reverse* and *__setitem__* which break conventions for the **SortedSet** class. So, we raise an exception whenever one of these methods is called.
- Now we present definition of the entire **SortedSet** class.

```
# Program: SortedSet1.py
# Authors: Michael H. Goldwasser
#          David Letscher
#
class SortedSet(list):
  """
  Maintains an ordered set of objects (without duplicates).
  """

  def __init__(self, initial=None):
    """
    Default constructor creates an empty SortedSet.

    If initial sequence is given as parameter, creates
    initial configuration using those elements, with
    duplicates removed.
    """
    list.__init__(self)   # calls the parent constructor
    if initial:
      self.extend(initial)

  def indexAfter(self, value):
    """
    Find first index of an element strictly larger than
    given value.

    If no element is greater than given value, this
    returns the length of the set.
    """
    walk = 0
    while walk < len(self) and value >= self[walk]:
      walk += 1
    return walk

  def insert(self, value):
    """
    Adds given element to the sorted set.

    If the value is already in the set, this has
    no effect. Otherwise, it is added in the proper
```

```
    location.

    value    element to be added to the set.
    """
    if value not in self:      # avoid duplicates
      place = self.indexAfter(value)
      list.insert(self, place, value) # the parent's method

  def append(self, object):
    """
    Identical to insert(object).
    """
    self.insert(object)

  def extend(self, other):
    """
    Extends the set by inserting elements from
    the other set
    """
    for element in other:
      self.insert(element)

  def __add__(self, other):
    """
    Returns new set which is union of this set
    and the other.
    """
    result = SortedSet(self)  # creates new copy of self
    result.extend(other)      # add other elements to this copy
    return result

  def sort(self):
    """
    This has no effect, as set is already sorted.
    """
    pass

  def reverse(self):
    """
    SortedSets cannot be reversed.
```

```
        This always raises a RuntimeError.
        """
        raise RuntimeError('SortedSet cannot be reversed')

    def __setitem__(self, index, object):
        """
        Direct manipulation of elements of a SortedSet
        is disallowed.

        This always raises a RuntimeError.
        """
        raise RuntimeError('This syntax not supported by SortedSet')
```

- **When should inheritance (not) be used**
- In the previous example, we noticed that some methods needed to be specialized (example, *insert*), some had to be ignored (example, *sort*), and some had to raise exceptions (example, *reverse, __setitem__*).
- There is an alternative way to take advantage of the existing **list** class without using inheritance.

```
class SortedSet:
    def __init__(self):
        self._items = list() #an initial empty list
```

- The relationship between a parent and child class when using inheritance is an **is-a relationship** in that every DeluxeTV is a Television.
- When one class is implemented using an instance variable of another, this is a **has-a relationship**.
- We are implementing a **SortedSet** has a **list** approach, even though **SortedSet** is itself not a **list**.
- Since we do not inherit from any parent class, we must explicitly provide support for any behaviors that we want to offer. For example,

```
def insert(self, value):
    if value not in self._items:
        place = self.indexAfter(value)
        self._items.insert(place,value)
```

- Note that in this version of *insert*, there is no concept of calling the "parent" version of insert in this context. Instead, we call *self._items.insert* to invoke the insert method of the underlying **list** attribute.

- The advantage of avoiding inheritance is that we do not inherit unnecessary baggage of methods. However, we do not inherit any of the necessary methods. Thus, we need to explicitly provide them. Here is the **SortedSet** class created without using inheritance.

```
# Program: SortedSet2.py
# Authors: Michael H. Goldwasser
#          David Letscher
#
#
#
class SortedSet:
  """Maintains an ordered set of objects (without duplicates)."""

  def __init__(self, initial=None):
    """Default constructor creates an empty SortedSet.

    If initial sequence is given as parameter, creates initial
    configuration using those elements, with duplicates removed.
    """
    self._items = list()
    if initial:
      self.extend(initial)       # extend the set (not the list)

  def indexAfter(self, value):
    """Find first index of an element strictly larger than given
    value.

    If no element is greater than given value, this returns the
    length of the set.
    """
    walk = 0
    while walk < len(self._items) and value >= self._items[walk]:
      walk += 1
    return walk

  def insert(self, value):
    """Adds given element to the sorted set.

    If the value is already in the set, this has no effect.
    Otherwise, it is added in the proper location.
```

```
    value    element to be added to the set.
    """
    if value not in self._items:
      place = self.indexAfter(value)
      self._items.insert(place, value)

  def extend(self, other):
    """Extends the set by inserting elements from the other set.
       """
    for element in other:
      self.insert(element)

  def __add__(self, other):
    """Returns new set which is union of this set and the other.
       """
    result = SortedSet(self)       # creates new copy of self
    result.extend(other)           # add other elements to this
       copy
    return result

  def index(self, value):
    """Returns index of value.

    Raises a ValueError if value not found.
    """
    return self._items.index(value)

  def remove(self, element):
    """Remove an element from the sorted set."""
    self._items.remove(element)

  def pop(self, index=None):
    """Pop element at given index (last by default)."""
    return self._items.pop(index)

  def __contains__(self, element):
    """Determine if the element is in the sorted set."""
    return element in self._items

  def __getitem__(self, index):
```

```
    """Get the element of the sorted set at the given index."""
    return self._items[index]

  def __len__(self):
    """Count the number of elements in the sorted set."""
    return len(self._items)

  def __eq__(self, other):
    """Determine if two sets are equivalent."""
    return self._items == other._items

  def __lt__(self, other):              # lexicographic comparison
    return self._items < other._items

  def __str__(self):
    """Return a string representation of the sorted set."""
    return str(self._items)
```

- **Class Hierarchies and cs1graphics**
- Suppose we want to draw stars,

FIGURE 9.5: Three instances of the Star class.

- We would like to be able to do the following:

```
medal = Star(5)
paper.add(medal)
```

- We use inheritance to avoid recreating everything from scratch.
- Start with

```
class Star(Polygon):
```

- Note, a star with n rays is a polygon with $2n$ vertices. The geometry of a star depends on both an outer radius, as well as an inner radius. We let the user specify the outer radius and the ratio between the inner and outer radii.

- Here is a code:

```
1  class Star(Polygon):
2    def __init__(self, numRays=5, outerRadius=10, innerRatio=.5, center=Point(0,0)):
3      Polygon.__init__(self)                  # call the parent constructor
4      top = Point(0, −outerRadius)            # top point is directly above the origin
5      angle = 180.0 / numRays
6
7      for i in range(numRays):
8        self.addPoint(top ^ (angle * (2 * i)))                    # outer point
9        self.addPoint(innerRatio * top ^ (angle * (2 * i + 1)))   # inner point
10
11     self.adjustReference(0, outerRadius)    # move reference from top point to center
12     self.move(center.getX( ), center.getY( ))  # re-center entire star
13     self._innerRatio = innerRatio           # record as an attribute
14
15   def setInnerRatio(self, newRatio):
16     factor = newRatio / self._innerRatio
17     self._innerRatio = newRatio
18     for i in range(1, self.getNumberOfPoints( ), 2):          # inner points only
19       self.setPoint(factor * self.getPoint(i), i)
```

FIGURE 9.6: Our proposed Star class (excluding documentation and unit testing).

- Here is a code to use inheritance to create a **Square** class as a child of a **Rectangle** class:

```
1  class Square(Rectangle):
2    def __init__(self, size=10, center=None):
3      Rectangle.__init__(self, size, size, center)
4
5    def setWidth(self, width):
6      self.setSize(width)
7
8    def setHeight(self, height):
9      self.setSize(height)
10
11   def setSize(self, size):
12     Rectangle.setWidth(self, size)          # parent version of this method
13     Rectangle.setHeight(self, size)         # parent version of this method
14
15   def getSize(self):
16     return self.getWidth( )
```

FIGURE 9.7: The actual implementation of the cs1graphics.Square class (documentation and error checking omitted).

- **DNA to RNA Transcription**
- Every organism consists of cells, all multicellular organisms have a cell and a cell nucleus.
- This nucleus contains the DNA, the hereditary material. This DNA is packed into chromosomes.
- DNA is short for Deoxyribonucleic acid. DNA is made up from 4 different bases (nucleotides): Adenine (A), Thymine (T), Guanine (G), and Cytosine (C).
- RNA (Ribonucleic Acid) is synthesized in the nucleus and is very similar to DNA. To make protein from DNA first RNA is made from DNA. This process is called **transcription**. The RNA is then used to create proteins.
- RNA also consists of four nucleotides, three of them A, C, and G, and a fourth one Uracil (U).
- Transcription creates an RNA sequence by matching a complementary base to each original base in the DNA using

DNA	→	RNA
A	→	U
C	→	G
G	→	C
T	→	A

- We develop a program that asks the user to enter a DNA sequence, and returns the transcribed RNA.
- So, a DNA sequence "AGGCTA" would be transcribed to "UCCGAU".

```
# Program: DNAtoRNA.py
# Authors: Michael H. Goldwasser
#          David Letscher

dnaCodes = 'ACGT'
rnaCodes = 'UGCA'

dna = input('Enter a DNA sequence: ')
rnaList = []
for base in dna:
  whichPair = dnaCodes.index(base)# index into dnaCodes
  rnaLetter = rnaCodes[whichPair] # corresp. index into rnaCodes
  rnaList.append(rnaLetter)
rna = ''.join(rnaList)              # join on empty string
print('Transcribed into RNA:', rna)
```

- **In what follows, we are using *cs1graphics.py*. The instructor will provide you with alternative TKinter files to build these classes.**
- **Drawing a Pyramid**
- We would like to draw pyramids (first the kind in (a) below and then (b)):

(a) (b)

FIGURE 4.6: Two versions of a pyramid. In (a) each level is a single rectangle; in (b) each level comprises a series of squares.

- The identifier *numLevels* stands for number of levels and *unitSize* represents the height of each level. Here is the geometric picture for what we want to do:

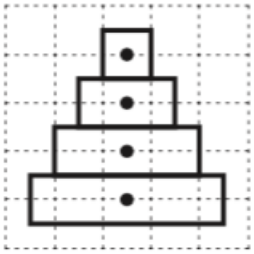


FIGURE 4.7: Geometric sketch for a 4-level pyramid. The dotted lines mark units in the coordinate system. The solid rectangles represent the levels of the pyramid, with each dot highlighting the desired center point for a level.

- Here is a code. Understand this loop:

```
# Program: pyramidLoop.py
# Authors: Michael H. Goldwasser
#          David Letscher
#
from cs1graphics import *

numLevels = 8                          # number of levels
unitSize = 12                          # the height of one level
screenSize = unitSize * (numLevels + 1)
paper = Canvas(screenSize, screenSize)
centerX = screenSize / 2.0             # same for all levels

# create levels from top to bottom
for level in range(numLevels):
```

```
  width = (level + 1) * unitSize     # width varies by level
  block = Rectangle(width, unitSize)# height is always unitSize
  centerY = (level + 1) * unitSize
  block.move(centerX, centerY)
  block.setFillColor('gray')
  paper.add(block)
```

- **Pyramid made of squares**
- Here, we use a nested loop to create and position a series of squares that comprise each level. Note, level k is comprised of $(k+1)$ squares. All the squares on a given level are centered with the same y coordinate.

```
# Program: pyramidNested.py
# Authors: Michael H. Goldwasser
#          David Letscher
#
#
from cs1graphics import *

numLevels = 8                  # number of levels
unitSize = 12                  # the height of one level
screenSize = unitSize * (numLevels + 1)
paper = Canvas(screenSize, screenSize)
centerX = screenSize / 2.0  # same for all levels

# create levels from top to bottom
for level in range(numLevels):
  # all blocks at this level have same y-coordinate
  centerY = (level + 1) * unitSize
  leftmostX = centerX - unitSize * level / 2.0
  for blockCount in range(level + 1):
    block = Square(unitSize)
    block.move(leftmostX + unitSize * blockCount, centerY)
    block.setFillColor('gray')
    paper.add(block)
```

- **Recursion – The Bull's Eye**
- Another form of repetition is **recursion.**
- **Structural recursion** is a natural way to define objects that have one or more (smaller) instances of the same class as instances. For instance, an 8-level pyramid is a single bottom level with a 7-level pyramid on top of it. A 7-level pyramid in turn is a single bottom level

with a 6-level pyramid on top of it. And so on, until a 1-level pyramid is a single block – this is the **base case**.

- **Functional recursion** occurs when a *behavior* is expressed using a smaller version of that same behavior. For instance, a function calls itself recursively.
- **A Bullseye Class**
- A Bull's eye is a sequence of concentric circles with alternating colors. In other words, a bull's eye is a circle with a smaller bull's eye positioned at the same center.

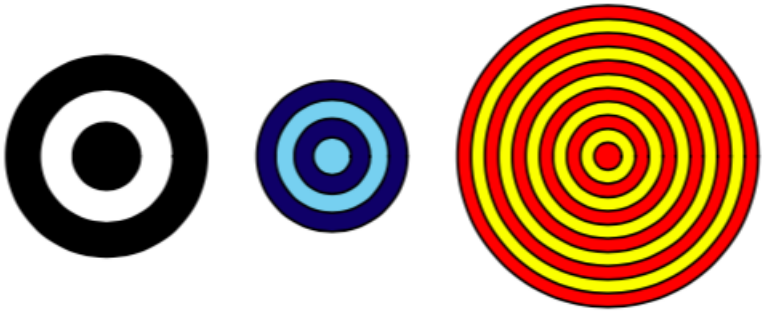


FIGURE 11.1: Bullseye instances with 3, 4, and 11 bands respectively.

- To draw a bull's eye we use structural recursion and develop a **Bullseye** class.
- Internally, a **Bullseye** instance maintains two attributes: _outer, which is the outermost circle, and _rest, which is a reference to another bullseye providing the interior structure. We allow the user to specify the total number of bands, the overall radius of the bullseye, and the choice of two colors.
- The constructor accepts four parameters (numBands, radius, primary, secondary) where the bands alternate in color starting with primary as the outermost band. The case case is self._rest is set to None.
- The entire implementation is given on the next page.

```
1  from cs1graphics import *
2
3  class Bullseye(Drawable):
4    def __init__(self, numBands, radius, primary='black', secondary='white'):
5      if numBands <= 0:
6        raise ValueError('Number of bands must be positive')
7      if radius <= 0:
8        raise ValueError('radius must be positive')
9
10     Drawable.__init__(self)                    # must call parent constructor
11     self._outer = Circle(radius)
12     self._outer.setFillColor(primary)
13
14     if numBands == 1:
15       self._rest = None
16     else:  # create new bullseye with one less band, reduced radius, and inverted colors
17       innerR = float(radius) * (numBands-1) / numBands
18       self._rest = Bullseye(numBands-1, innerR, secondary, primary)
19
20   def getNumBands(self):
21     bandcount = 1                              # outer is always there
22     if self._rest:                             # still more
23       bandcount += self._rest.getNumBands( )
24     return bandcount
25
26   def getRadius(self):
27     return self._outer.getRadius( )            # ask the circle
28
29   def setColors(self, primary, secondary):
30     self._outer.setFillColor(primary)
31     if self._rest:
32       self._rest.setColors(secondary, primary) # color inversion
33
34   def _draw(self):
35     self._beginDraw( )                         # required protocol for Drawable
36     self._outer._draw( )                       # draw the circle
37     if self._rest:
38       self._rest._draw( )                      # recursively draw the rest
39     self._completeDraw( )                      # required protocol for Drawable
```

FIGURE 11.2: Complete code for our Bullseye implementation (documentation excluded).

- **Unfolding a recursion**
- Here is a figure to portray the creation of a bullseye instantiated as *Bullseye(3,120, 'red', 'black')*

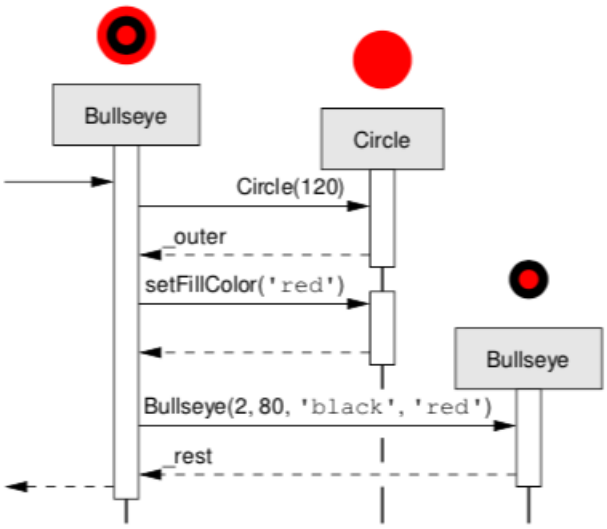


FIGURE 11.3: Top-level trace of Bullseye(3, 120, 'red', 'black')

- To truly understand recursion, it helps to more carefully trace the complete execution through a process called **unfolding a recursion.**
- The next page gives the complete execution. Note, the last case is traced as well, but the construction algorithm proceeds differently because the desired number of bands is one. An appropriate outer circle is constructed and colored, but the rest of this bullseye is set to **None**.

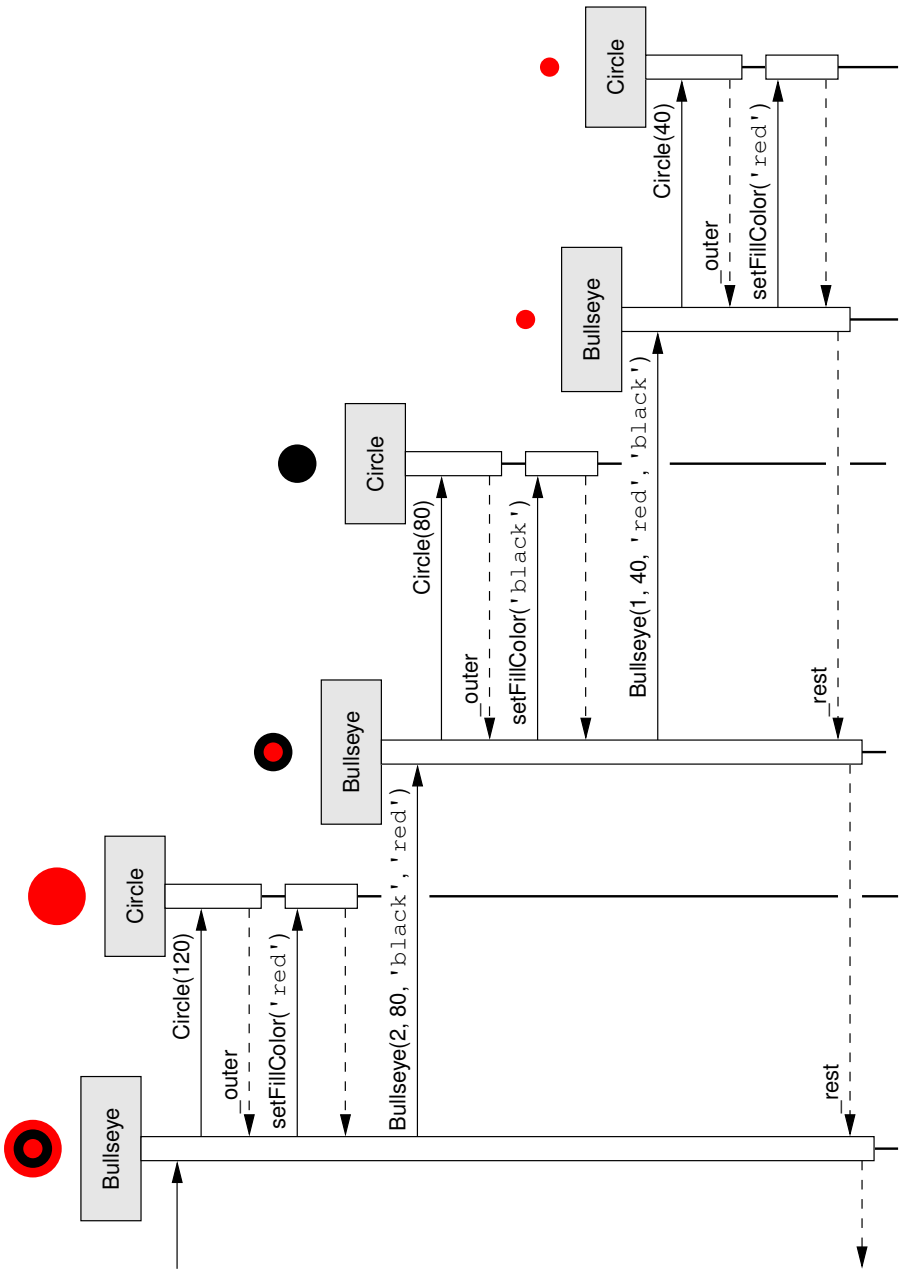


FIGURE 11.4: Unfolding the recursion Bullseye(3, 120, `'red'`, `'black'`)

- **Additional Bullseye methods**
- Our class supports several additional methods, *getNumBands* and *getRadius* method. Note, *getRadius* is not recursive; rather this invokes the *getRadius* method of the **Circle** instance (not of another **Bullseye** instance). This is an example of polymorphism, as both classes support a *getRadius* method with different underlying implementations.
- The mutator *setColors(primary, secondary)* recolors an existing bullseye.
- Having inherited from **Drawable**, our bullseye automatically supports behaviors such as *move, scale, rotate, clone.*
- But, we are responsible for implementing the *_draw* method to display the bullseye.
- Here is a sequence diagram:

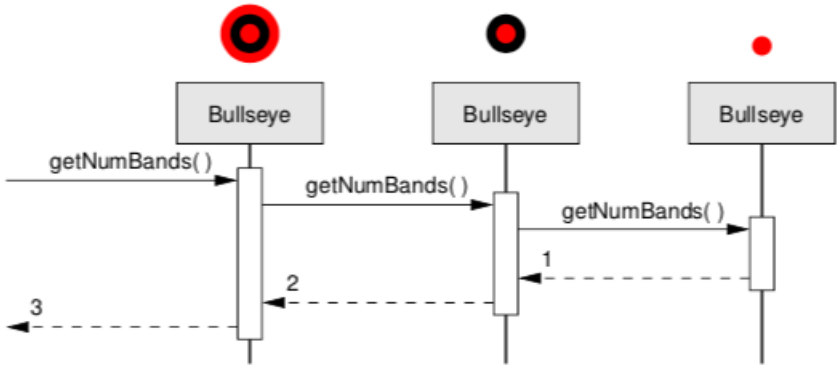


FIGURE 11.5: Sequence diagram for the call getNumBands( ).

- **Finally, a word of caution:** Every recursion must have a base case.

- **Functional Recursion**
- Recall, the factorial of a number: $n! = n \cdot (n-1) \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$
- The number of ways that n items can be ordered (permutations) is $n!$
- Note, $n! = n \cdot (n-1)!$. This calls for a recursion (although, a loop could also be easily used).

```
def factorial(n):
  """Compute the factorial of n.

  n is presumed to be a positive integer.
  """
  if n <= 1:
    return 1
  else:
    return n * factorial(n-1)
```

- Each time a function is called, the system creates an activation record to track the state of that particular invocation. In the context of recursion, each individual call to the function executes the same body of code, yet with a separate activation record.
- Here is a figure to explain the recursion involved in *factorial(4)*:

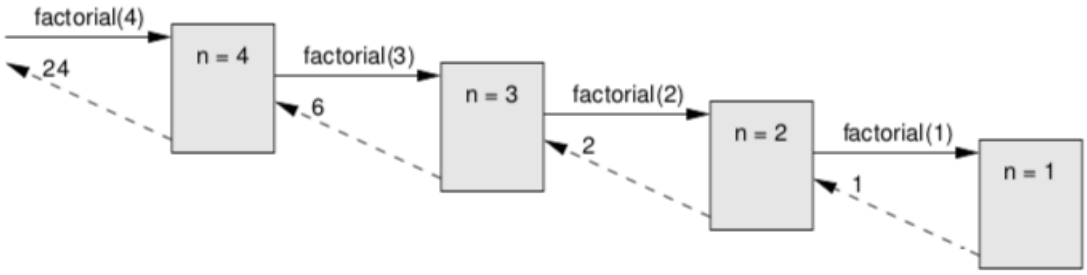


FIGURE 11.12: The trace of a call to factorial(4).

- **Binary Search**
- We want to know whether a specific value occurs in a list.
- Python's **list** class uses a **sequential search.** That is, the program starts scanning from the beginning of the list until it either finds what it is looking for, or reaches the end of the list and reports a failure.
- Assume that the list of values is *sorted*. For example, a **lexicon** is a list of strings. It is preferable to maintain a sorted list.
- **Algorithmic Design**
- In binary search we go through a sorted list by aiming right at the middle. If our target value is not right in the middle, then it is in one of the two sub-lists; one of the left and one on the right. In other words, the search now proceeds to one of the half-lists and we repeat with recursion.
- The advantage of the binary search in comparison to the sequential list is that the binary search is much faster. Using binary search on a list of length n, we obtain a result within $\log_2(n)$ number of steps. Whereas, the sequential search can take-up to n steps.

- Here is an implementation.

```
1  def search(lexicon, target):
2    """Search for the target within lexicon.
3
4    lexicon     a list of words (presumed to be alphabetized)
5    target      the desired word
6    """
7    if len(lexicon) == 0:                              # base case
8      return False
9    else:
10     midIndex = len(lexicon) // 2
11     if target == lexicon[midIndex]:                  # found it
12       return True
13     elif target < lexicon[midIndex]:                 # check left side
14       return search(lexicon[ : midIndex], target)
15     else:                                            # check right side
16       return search(lexicon[midIndex+1 : ], target)
```

FIGURE 11.13: A bad implementation of the binary search algorithm.

- Here is a diagram of an example: *searc(['B','E','G','I','N','S'], 'F')* :

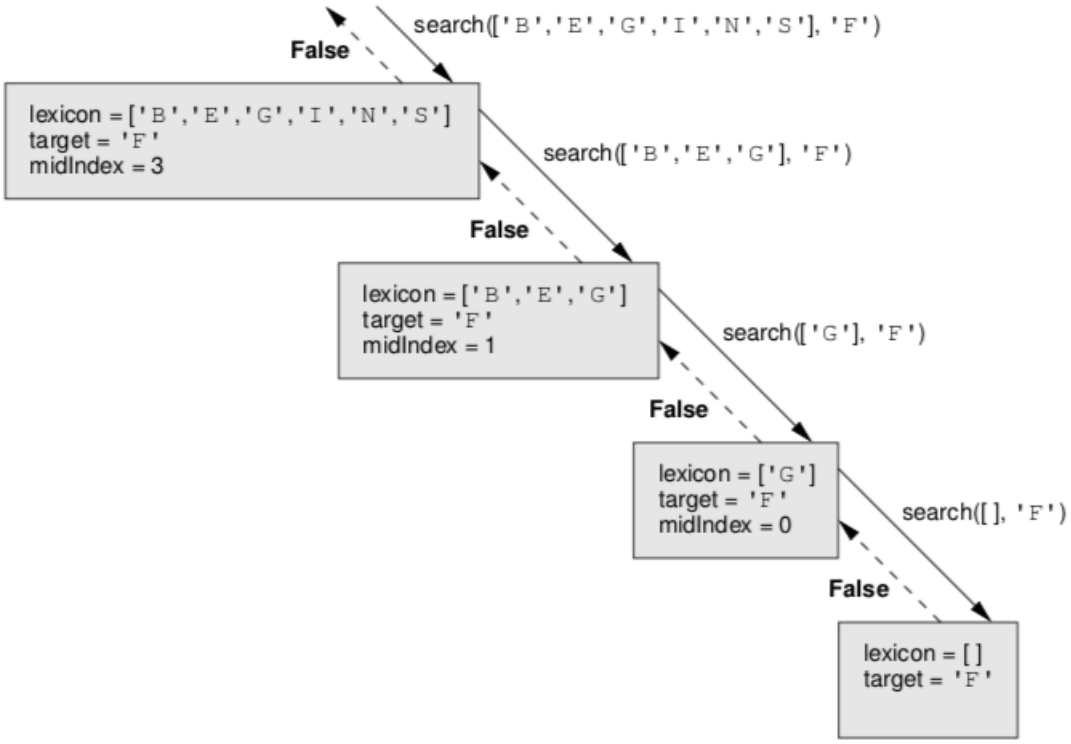


FIGURE 11.14: The trace of the call search(['B','E','G','I','N','S'], 'F') for the inferior implementation of binary search from Figure 11.13.

- This is not an ideal implementation because the recursion relies on creating a **new** list which is half the original list. Creating this slice takes time proportional to the length of the slice, and thus ruining the benefits of the binary search algorithm.

- Here is a better implementation – note, we do not create new sublists.

```
1  def search(lexicon, target, start=0, stop=None):
2    """Search for the target within lexicon[start:stop].
3
4    lexicon     a list of words (presumed to be alphabetized)
5    target      the desired word
6    start       the smallest index at which to look (default 0)
7    stop        the index before which to stop (default len(lexicon) )
8    """
9    if stop is None:
10     stop = len(lexicon)
11   if start >= stop:                                  # nothing left
12     return False
13   else:
14     midIndex = (start + stop) // 2
15     if target == lexicon[midIndex]:                  # found it
16       return True
17     elif target < lexicon[midIndex]:                 # check left side
18       return search(lexicon, target, start, midIndex)
19     else:                                            # check right side
20       return search(lexicon, target, midIndex+1, stop)
```

FIGURE 11.15: Preferred implementation of the binary search algorithm.

- Passing a reference to the list is quite efficient, as it does not involve copying the object. We use indices *start* and *stop*. Note, the idea is the same as in the previous case, but we save time by simply not copying sublists. Here is a diagram for the same example, *searc(lexicon, 'F')* where *lexicon = ['B','E','G','I','N','S']*:

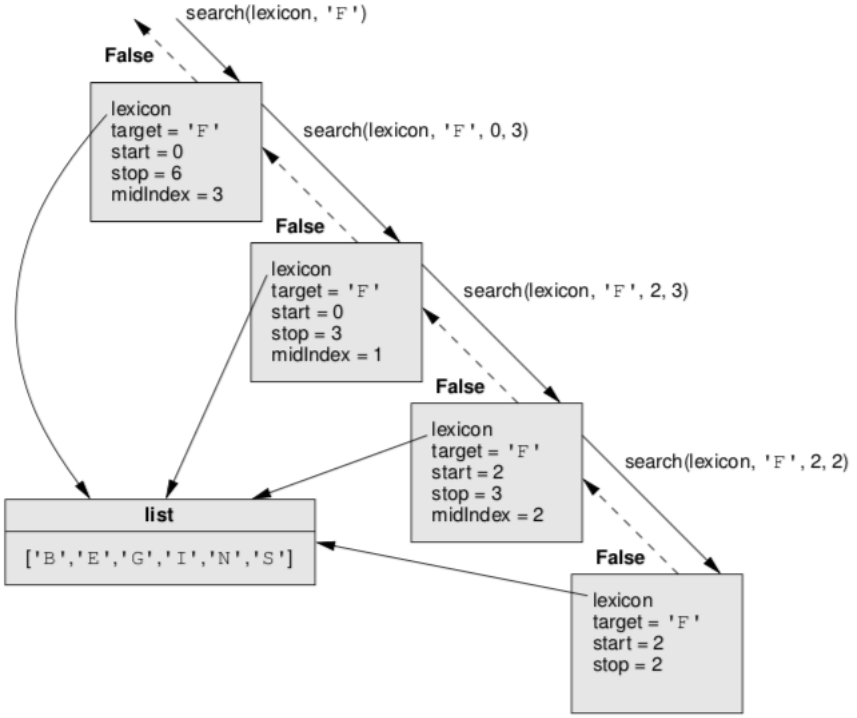


FIGURE 11.16: The trace of the call, search(['B','E','G','I','N','S'], 'F') for the improved implementation of binary search from Figure 11.15.

- Sometimes we are not looking for an exact match of our target in the given list. Sometimes we want to just match the first letter (in case of word search) or first digit (in case of number search):
- Here is an implementation of the *prefixSearch*:

```
1  def prefixSearch(lexicon, target, start=0, stop=None):
2    """Search to see if target occurs as a prefix of a word in lexicon[start:stop].
3
4    lexicon     a list of words (presumed to be alphabetized)
5    target      the desired word
6    start       the smallest index at which to look (default 0)
7    stop        the index before which to stop (default len(lexicon) )
8    """
9    if stop is None:
10     stop = len(lexicon)
11   if start >= stop:
12     return False
13   else:
14     midIndex = (start + stop)//2
15     if lexicon[midIndex].startswith(target):                    # found prefix
16       return True
17     elif target < lexicon[midIndex]:                            # check left side
18       return prefixSearch(lexicon, target, start, midIndex)
19     else:                                                       # check right side
20       return prefixSearch(lexicon, target, midIndex+1, stop)
```

FIGURE 11.17: Using binary search to check whether a prefix occurs within a list of words.

16. Container Classes (list vs. tuple; Dictionary, Set, Frozen Set (sections 12.1, 12.2, 12.3)

- Python has several classes that provide support for managing a collection. Example, **list, tuple.** We call any such object a **container.**
- By design, Python's containers all support certain common syntaxes. Example, *for element in data*, *element in data*, *len(data)*.
- There are differences in the behaviors and efficiencies of operations in the various container classes.
- We first look at examples. Try out the following:

```
##################################################
##################################################
#Lists
##################################################
##################################################

list1 = list() #Constructor
list1.append('mathematics')
list1.append(10)
list1.append([])
list1.append((4,5,6))
print(list1)

#List is ordered, mutable, and heterogeneous

list2 = ['physics', (2,3,5), [20,30], 256] #Literal form
##################################################
##################################################
#Tuples
##################################################
##################################################

tuple1 = tuple() #Constructor
tuple1.append('mathematics') #Will give AttributeError
    #Tuples are not mutable.
tuple2 = ('mathematics', 24, [1,2], (3,4,5))
                #Literal form



#Tuple is ordered and heterogeneous.
#Tuples are not mutable.
```

```
##################################################
##################################################
#Dictionary
##################################################
##################################################


info = dict() #Constructor
info['Name'] = 'Jane'
info['Age'] = 11
info['Grade'] = 5

print(info)

fruits = {'sweet': 'mango', 'bitter':'lemon'}
        #Literal form

#Unlike lists or tuples, the keys in a dictionary can
#be non-numeric.
#Dictionaries are mutable, associative, and heterogeneous.
#Dictionaries are not ordered.


##################################################
##################################################
#Set
##################################################
##################################################

s = set() #Constructor
s.add(32)
s.add(42)
s.add(3)
s.add('abc')
print(s)

t = {'a', 23,45} #Literal
print(t)
```

```
#Set is unordered.
#Set is mutable, heterogeneous.
#Set is not associative.

##############################################
##############################################
#Frozen Set
##############################################
##############################################

f1 = frozenset() #Constructor.  Cannot be mutated.

f2 = frozenset([3,4,'abc'])
    #f2 created from a list

#The only way to create a frozenset is by using
#an iterable parameter (list, set, tuple, etc.).

#Frozenset is not mutable (analogue of a tuple).
#Frozenset is not ordered, not associative (analogue of a set).
#Frozenset is heterogeneous.


##############################################
##############################################
#Array
##############################################
##############################################

#Arrays are meant to be homogeneous
#We need to import the array module to be able to
#build an array.

import array

a1 = array.array('d', [1,3.4, 5.5])
print(a1)
```

Commonly used **type** codes:

Code	Type	Minimum size **in** bytes

```
'b'         int                 1
'B'         int                 1
'u'         Unicode character   2
'h'         int                 2
'H'         int                 2
'i'         int                 2
'I'         int                 2
'l'         int                 4
'L'         int                 4
'f'         float               4
'd'         float               8

#Arrays are ordered and mutable.
```

- Here are some aspects of any container:
- **order** – The classes **list, tuple, array** are used to represent an ordered sequence of elements. They use the concept of index for designing the location of an element within the sequence.
- **mutability** – The distinction between a **list** and a **tupe** is that the list is mutable whereas the tuple is immutable. So we can insert, replace, or remove elements of a list, which we cannot do with a tuple.
- **associativity** – Python's **dict** (dictionary) class is used to represent an association between a **key** and its **value**.
- **heterogeneity** – A collection is called **heterogeneous** if it can hold elements of different types. If a collection can hold only one type of elements, then that collection is called **homogeneous.** Most of Python's containers support heterogeneity.
- **storage** – Most of Python's containers are referential, in that the value of the elements are not stored internal to the container, but indirectly as references to other objects. This is the reason that heterogeneity is supported. But the indirect nature of the storage can be less efficient in terms of memory usage and access time. For high-performance applications, Python supports an **array** class, which provides compact storage for a collection of data drawn from a chosen primitive type.
- Here is a summary:

	list	tuple	dict	set	frozenset	array
ordered	✓	✓				✓
mutable	✓		✓	✓		✓
associative			✓			
heterogeneous	✓	✓	✓	✓	✓	
compact storage						✓

FIGURE 12.1: A summary of the varying aspects of Python's containers.

- **Two Familiar Containers: list and tuple**
- The classes **list** and **tuple** are used to manage an *ordered sequence* of elements. The position of a particular element within that sequence is designated with an **index**.
- We may view both the **list** and **tuple** as an association between indices and values. Note, the indices must be consequent integers from 0 to one less than the length of the sequence.
- **Limitations to the Use of a list or tuple**
- **Lack of permanency**
- The elements in a list are identified using indices. But if one element is removed, then indices of all the elements which come after changed. For instance, in a list named *employee*, say *employee[15]* retires, then the employees previously numbered 16, 17, 18, ... are now numbered 15, 16, 17, ... In other words, This means, an index cannot serve as a meaningful and permanent identifier for an element.
- **The problem of large integers**
- Suppose we want to identify employees by their social security numbers, or identify books using their 13-digit ISBN numbers. Placing these elements in a long enough list to accommodate all the social security numbers or ISBN numbers is impractical, since we are interested only in a small fraction of such a long list.
- **Non-numerc identification**
- There are elements which do not have numbering scheme. For example, movies do not have ISBN or other numbering schemes. Or countries, their capital cities, or cities and their populations. Instead of a **list**, it would be much better to use a **dict** class
- **Dictionaries**
- A dictionary is an associative container. It represents a mapping from objects known as **keys** to associated objects known as **values**. Unlike lists or tuples, in the dictionaries keys can be non-numeric.
- **The keys:** A key serves as an identifier when accessing a particular value in the dictionary. They keys within a dictionary are required to be unique. Thus, employers rely on social security numbers of their employees, or books are numbered using their unique ISBNs. Note, keys could themselves be tuples so as to make certain that they are unique. For instance, the first name of students may not suffice to distinguish students. It is better to use the tuple (firstname, lastname) as a key.
- Keys must be drawn from an *immutable class*, such as **int, str,** or **tuple**. The elements of a dictionary are not inherently ordered as with a list. If a key were allowed to change, the original placement of the element may no longer match the expected placement based on the updated key.
- In other words, **all keys within a given dictionary much be unique and immutable.**
- **The values**
- There are no restrictions on the allowable values in a dictionary. They can be from any class, and they are not required to be unique.
- **Python's dict Class:** Here is an example of building a dictionary

```
flowers = dict()  #Standard constructor syntax. Or,
flowers = {} #Literal form
```

```
flowers['red'] = 'rose'
flowers['yellow'] = 'daffodil'
flowers['white'] = 'lilly'
```

- Here is another dictionary:

```
DnaToRna = {'A': 'U', 'C': 'G', 'G': 'C', 'T': 'A'}
```

- **Supported behaviors**
- Here are some selected supported behaviors. Build a dictionary of your choice either using a constructor or in a literal approach, and implement every method in the list below.

Syntax	Semantics
d[k]	Returns the item associated with key k; results in KeyError if k not found.
d[k] = value	Associates the key k with the given value.
k **in** d	Returns **True** if dictionary contains the key k; **False** otherwise.
k **not in** d	Returns **True** if dictionary does not contain the key k; **False** otherwise.
len(d)	Returns the number of (key,value) pairs in the dictionary.
d.clear( )	Removes all entries from the dictionary.
d.pop(k)	Removes key k and its associated value from dictionary, returning that value to the caller; raises a KeyError if not found.
d.popitem( )	Removes and returns an arbitrary (key, value) pair as a tuple.
d.keys( )	Returns a **list** of all keys in the dictionary (in arbitrary order).
d.values( )	Returns a **list** of all associated values in the dictionary (in arbitrary order).
d.items( )	Returns a **list** of tuples representing the (key, value) pairs in the dictionary.
for k **in** d**:**	Iterates over all keys of the dictionary (in arbitrary order); this shorthand is equivalent to **for** k **in** d.keys( )**:**.

FIGURE 12.2: Selected behaviors of Python's **dict** class, for a prototypical instance d.

- **Containers of Containers**
- One may have a **list, tuple, dict** contain any type of object. In particular, one may have a list of lists, a tuple of dictionaries, a dictionary of lists etc.,.
- **Modeling Multidimensional Tables**
- Here is a Tic-tac-toe board written in the form of a list of lists:

```
game = [['X', 'X', 'O'], ['O', 'O', 'X'], ['X', 'O', 'X']]
```

- Storing a two-dimensional data set in this way is called **row-major** order:

X	X	O
O	O	X
X	O	X

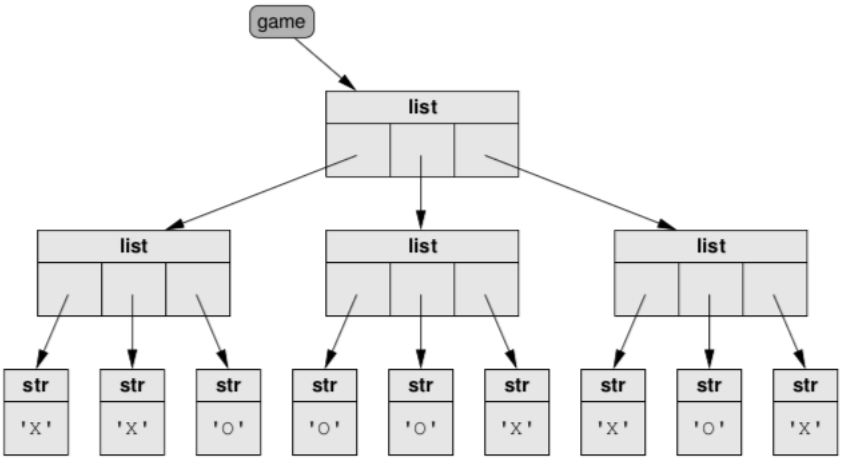


FIGURE 12.3: A diagram of the Tic-tac-toe representation as a list of rows, which themselves are lists of strings.

- Note, *game[i][j]* returns the entry on the i-th row and j-th column.
- We may also have built *game* in a **column-major** order, as a list of columns.
- **Modeling Many-to-Many Relationships**
- Recall that a dictionary is a **many-to-one**relationship between keys and values.
- Distinct keys could very well map to a single value, but one key cannot be mapped to many values.
- Suppose we build a dictionary with a **many-to-many** relationship. That is, one key gets mapped to a tuple.
- For instance, suppose we build a dictionary *flowers* where we have *flowers['red'] = ('rose', 'hibiscus')* and several such examples.
- Then *'rose' in flowers* returns False since 'red' is not a key.
- Likewise *'rose' in flowers.values()* is also False, since *flowers.values()* is a list of tuples.
- The command *'rose' in flowers['red']* returns True.
- **Reverse Dictionary**
- As we know, a dictionary is a many-to-one relationship. Therefore, a **reverse dictionary** would be a one-to-many relationship. Here is a function to build a reverse-dictionary from a dictionary. Understand and test it.

```
def buildReverse(dictionary):
    """Return a reverse dictionary based upon the original."""
    reverse = {}
    for key,value in dictionary.items(): #map value back to key
        if value in reverse:
            reverse[value].append(key)  #add to existing list
        else:
            reverse[value] = [key]      #establish new list
    return reverse
```

- **12.4: Sets**
- A set is an unordered collection of unique elements.

- Python has two new built-in classes: **set** and **frozenset**. They both model the concept of a mathematical set, with one distinction: **set** is mutable while **frozenset** is immutable.
- Note though, elements added to a set must be immutable.
- **Constructor:** The syntax for constructing a set is: *set()*, which by default produces an empty set.
- There is no literal form for a set, but a set can be formed from a list, string, tuple, or dictionary.
- Example, *set([1,2,3,4])*, or *set('aeiou')*, or *set((1,2,3))*.
- Similarly, one may build a **frozenset** from a list by using the syntax, *frozenset(list)*.
- Sets and Frozensets support accessors listed here:

Syntax (with alternate)	Semantics
len(s)	Returns the cardinality of set s.
v **in** s	Returns **True** if set s contains the value v, **False** otherwise.
v **not in** s	Returns **True** if set s does not contain the value v, **False** otherwise.
for v **in** s:	Iterates over all values in set s (in arbitrary order).
s == t	Returns **True** if set s and set t have identical contents, **False** otherwise (order is irrelevant).
s < t	Returns **True** if set s is a *proper* subset of t, **False** otherwise.
s <= t s.issubset(t)	Returns **True** if set s is a subset of t, **False** otherwise.
s > t	Returns **True** if set s is a *proper* superset of t, **False** otherwise.
s >= t s.issuperset(t)	Returns **True** if set s is a superset of t, **False** otherwise.
s | t s.union(t)	Returns a new set of all elements that are in either set s or set t (or both).
s & t s.intersection(t)	Returns a new set of all elements that are in both set s and set t.
s − t s.difference(t)	Returns a new set of all elements that are in set s but not in set t.
s ^ t s.symmetric_difference(t)	Returns a new set of all elements that are in either set s or set t but not both.

FIGURE 12.4: Accessor methods supported by the **set** and **frozenset**.

- Here are methods which mutate sets.

Syntax (with alternate)	Semantics
s.add(v)	Adds value v to the set s; has no effect if already present.
s.discard(v)	Removes value v from the set s if present; has no effect otherwise.
s.remove(v)	Removes value v from the set s if present; raises a KeyError otherwise.
s.pop( )	Removes and returns arbitrary value from set s.
s.clear( )	Removes all entries from the set s.
s |= t s.update(t)	Alters set s, *adding* all elements from set t; thus set s becomes the union of the original two.
s &= t s.intersection_update(t)	Alters set s, *removing* elements that are not in set t; set s becomes the intersection of the original two.
s −= t s.difference_update(t)	Alters set s, *removing* elements that are found in set t; set s becomes the difference of the original two.
s ^= t s.symmetric_difference_update(t)	Alters set s to include only elements originally in exactly one of the two sets, but not both.

FIGURE 12.5: Behaviors that mutate a **set**.

- Some of these mutators can be applied to frozen sets. The intersection (or union) of frozensets results in a frozenset.
- Check out which of the mutators can be applied to frozensets. Use examples.
- What happens when we consider intersection of a set with a frozenset? Order matters.

17. Event-driven programming, graphics module, event handling (sections 15.1, 15.2, 15.3, 15.4)

- An **event-driven programming** is a paradigm where an executing program waits passively for external **events** to occur and then responds appropriately to those events.
- We focus primarily on **graphical user interfaces (GUIs)**.
- **Basics of Event-Driven Programming**
- Here is a basic example of event-driven programming:

```
name = input("What is your name? ") #Wait for response from user
print("Hello %s. Nice to meet you." %name)
```

- Here is another example:

```
paper = Canvas()
cue = paper.wait() #Wait for response from user
ball = Circle(10, cue.getMouseLocation())
ball.setFillColor('red')
paper.add(ball)
```

- Both these examples are sequential. In most software, the user has more freedom to control the actions of a program. At any point, a user has the option of selecting menu items, entering keyboard input, using a scroll bar, selecting text, and much more.
- **Event handling** is an approach by which the program is able to declare various events that should be available to the user, and then provides explicit code that should be followed to handle each individual type of event when triggered. This piece of code is known as an **event handler.**
- **Event handlers**
- Often a separate event handler is declared for each kind of event that can be triggered by a user interaction.
- An event handler is typically implemented either as a stand-alone function or as an instance of a specially defined class.
- When programmed as a stand-alone function, event handlers are known as **callback functions.**
- The appropriate callback function is registered in advance as a handler for a particular kind of event. This is sometimes described as registering to **listen** for an event, and thus handlers are sometimes called **listeners.** Each time such an event subsequently occurs, this function will be called.
- With object-oriented programming, event handling is typically implemented through an event-handling class. An instance of such a class supports one or more member functions that will be called when an appropriate event occurs. The advantage of this technique over use of pure callback function is that a handler can maintain state information to coordinate the responses for a series of events. For instance, *wait()* is supported by cs1graphics module.
- **The event loop**

- The design of event-driven software is quite different from our traditional flow-driven programming, although there is still a concept of the main flow of control. When the software first executes, initialization is performed in traditional fashion, perhaps to create and decorate one or more windows and set up appropriate menus. It is during this initialization that event handlers are declared and registered.
- We want our software to handle any number of predefined events triggered in arbitrary order.This is accomplished by having the main flow of control enter what is known as an **event loop.** This is essentially an infinite loop that does nothing.
- When an event occurs, the loop stops to look for an appropriately registered handler, and if found that handler is called (otherwise the event is ignored).
- When a handler is called, the flow of control is temporarily ceded to the handler, which responds appropriately. Once the handler completes its task, the default continuation is to re-enter the event loop. If the appropriate consequence of a user action is to "quit' the event loop.
- For example, every time we use the cs1graphics package, an event loop runs concurrently with the rest of our program. Every time we click on the canvas' window or typed on the keyboard the event loop was informed. One exception is when we click on the icon to close the window; the event loop triggers a built-in handler that closes the window. When all canvas windows are closed, the event loop terminates. This starting and stopping of the event loop happens automatically.

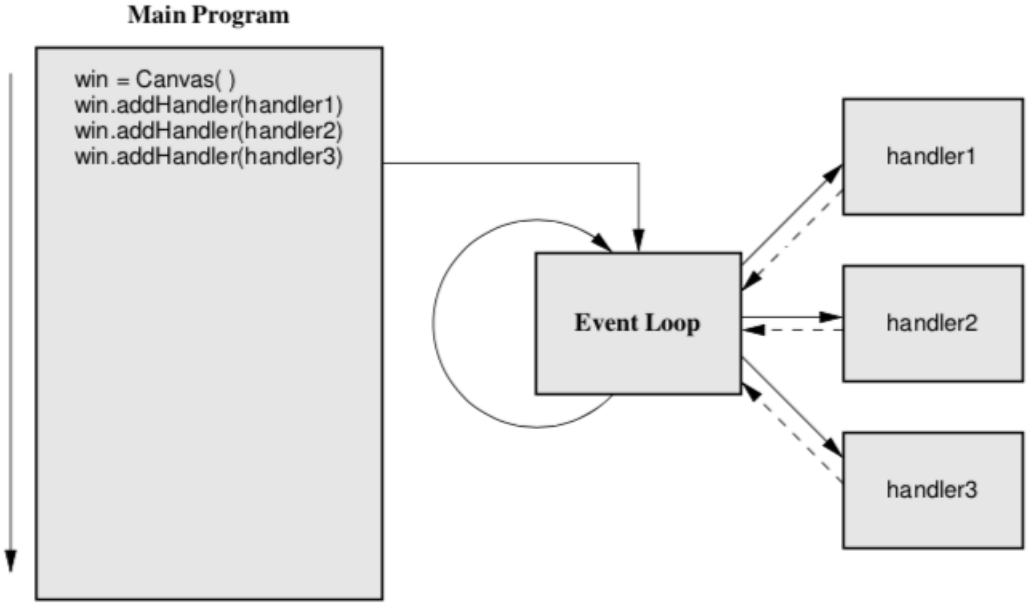


FIGURE 15.1: The flow of control in an event-driven program.

- **Threading**
- There are several forms of event-driven programming. The first question at hand is what happens to the flow of control once an event loop begins. In one model, the primary program cedes the flow of control to the event loop
- Another model uses what is known as **multithreaded** programming, allowing the main program to continue executing even while the event loop is monitoring and responding to events.

- The main routine and this event loop run simultaneously as separate **threads** of the program. Threading can be supported by the programming language and the underlying operating system. In reality the threads are sharing the CPU each given small alternating time slices in which to execute.
- **What follows heavily depends on *cs1graphics.py*. Alternatively, you may work with codes written with TKinter available in the appendix.**
- **The EventHandler Class**
- The *cs1graphics* module includes a generic **EventHandler** class that should be used as a parent class when defining your own handlers. The two methods of this class are the constructor itself and a *handle* mathod.

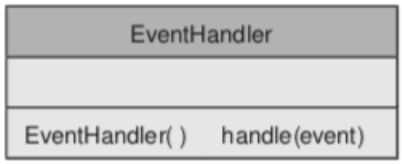


FIGURE 15.2: The EventHandler class.

- The *handle* method does not itself do anything. This method is overridden by a child class to describe the proper action in case of an event. The parameter to the *handle* method is an instance of an **Event** class, used to describe the particular event that occurred. For example,

```
class BasicHandler(EventHandler):
    def handle(self, event):
        print('Event Triggered')
```

- **Registering a handler with a graphics object**
- For an event handler to be active, it must be registered with one or more graphical objects.
- Both the **Canvas** and **Drawable** classes support two additional methods named *addHandler* and *removeHandler* to register and unregister a handler. For instance,

```
simple = BasicHandler()
paper=Canvas()
paper.addHandler(simple)
```

- Using a similar technique we can register an event handler directly with a particular drawable object:

```
sun = Circle(30, Point(50,50))
sun.setFillColor('yellow')
paper = Canvas()
paper.add(sun)
simple = BasicHandler()
sun.addHandler(simple)
```

- Now the handler can be trigged only when an event is received by the sun. Clicking on any other part of the canvas will not suffice.

- Note, it is possible to register a single handler with multiple shapes or canvases, and to have multiple handlers registered with the same shape or canvas. If there are multiple handlers registered with an object and a relevant event is triggered then each of those handlers is called in the order that they were registered.
- **A Handler with State Information**
- Here is an example of a handler that counts the number of times that it has been triggered.

```
# Program: CountingHandler.py
# Authors: Michael H. Goldwasser
#          David Letscher
#
from cs1graphics import *

class CountingHandler(EventHandler):
  def __init__(self):
    EventHandler.__init__(self)   # call the parent constructor!
    self._count = 0

  def handle(self, event):
    self._count += 1
    print('Event Triggered. Count:', self._count)
```

- Here is another example that updates a graphical count in the form of a **Text** object.

```
# Program: TallyHandler.py
# Authors: Michael H. Goldwasser
#          David Letscher
#
#
from cs1graphics import *

class TallyHandler(EventHandler):
  def __init__(self, textObj):
    EventHandler.__init__(self)
    self._count = 0
    self._text = textObj
    self._text.setMessage(str(self._count))  # reset to 0

  def handle(self, event):
    self._count += 1
    self._text.setMessage(str(self._count))
```

- **The Event Class**

- Sometimes, an event handler may need information about the triggering event, such as the mouse location or the type of event that was received. So, we pass an additional parameter

```
def handle(self, event):  #Here, event is an instance of an Event
    class
```

- The **Event** class supports a method *getTrigger()* which returns a reference to the underlying object upon which the event was originally triggered (a canvas or a drawable object). Using this information, we can rewrite the class **HandleOnce** as follows:

```
class HandleOnce(EventHandler):
    def handle(self, event):
        print("That's all folks!!!")
        event.getTrigger().removeHandler(self)

paper = Canvas()
oneTime = HandleOnce()
paper.addHandler(oneTime)
```

- The **Event** class also supports a *getDescription()* accessor that returns a string indicating the kind of event that occurred.
- **Mouse Events**
- Mouse events can be recognized by *getDescription()* method which returns a string, 'mouse click', or 'mouse release,' or 'mouse drag.' For instance, when the user clicks the mouse and leaves the mouse, it results in two mouse events: 'mouse click' and 'mouse release.'
- The precise position of the mouse is given by *getMouseLocation()*. This method of the **Event** class returns a **Point** instance. Here is a program that uses these methods.

```
class CircleDrawHandler(EventHandler):
    def handle(self, event):
        if event.getDescription() == 'mouse click':
            c = Circle(5, event.getMouseLocation())
            event.getTrigger().add(c)

paper = Canvas(100,100)
handler = CircleDrawHandler()
paper.addHandler(handler)
```

- Notice that this handler ignores other kinds of events. Try changing the code so that it draws the circle on a *mouse release*.
- The method *getOldMouseLocation()* indicates where the mouse was before a 'mouse drag' event was implemented.
- Here is a program to understand 'mouse drag' event.

```
# Program: ClickAndReleaseHandler.py
# Authors: Michael H. Goldwasser
```

```
#           David Letscher
#
#
from cs1graphics import *

class ClickAndReleaseHandler(EventHandler):
  def __init__(self):
    EventHandler.__init__(self)
    self._mouseDragged = False

  def handle(self, event):
    if event.getDescription() == 'mouse click':
      self._mouseDragged = False
    elif event.getDescription() == 'mouse drag':
      self._mouseDragged = True
    elif event.getDescription() == 'mouse release':
      if self._mouseDragged:
        print('Mouse was dragged')
      else:
        print('Mouse was clicked without dragging')

if __name__ == '__main__':
  paper = Canvas()
  dragDetector = ClickAndReleaseHandler()
  paper.addHandler(dragDetector)
```

- **Keyboard Events**
- When a keyboard is used, the *getDescription()* reports 'keyboard.' If needed, the *getMouse-Location()* is supported for a keyboard event.
- Keyboard events also support a behavior *getKey()* that returns the single character that was typed on the keyboard to trigger the event.
- Note that if the user types a series of characters, each one of those triggers a separate event.

- Here is a program showing how to display characters graphically as they are typed within a canvas.

```
# Program: KeyHandler.py
# Authors: Michael H. Goldwasser
#          David Letscher
#
from cs1graphics import *

class KeyHandler(EventHandler):
  def __init__(self, textObj):
    EventHandler.__init__(self)
    self._textObj = textObj

  def handle(self, event):
    if event.getDescription() == 'keyboard':
      self._textObj.setMessage(self._textObj.getMessage() + event.
          getKey())
    elif event.getDescription() == 'mouse click':
      self._textObj.setMessage('')   # clear the text

if __name__ == '__main__':
  paper = Canvas()
  textDisplay = Text('', 12, Point(100,100))  # empty string
  paper.add(textDisplay)
  echo = KeyHandler(textDisplay)
  paper.addHandler(echo)
```

- **Timers**
- Other than user triggered activities involving mouse and keyboard, there are other events generated internally.
- The *cs1graphics* module includes the **Timer** class. A timer instance is a self-standing object that is used to generate new events. The syntax *Timer()* creates a timer that generates an event after ten seconds has elapsed. The *start()* method must be called to begin the timer's clock.
- For a timer to be useful, there must be a corresponding handler that is registered to listen for those events.
- Here is an example of animating a rotating shape:

```
class RotationHandler(EventHandler):
  def __init__(self, shape):
    self._shape = shape

  def handle(self, event):
    self._shape.rotate(1)

paper = Canvas(100,100)
sampleCircle = Circle(20, Point(50,20))
sampleCircle.adjustReference(0,30)
paper.add(sampleCircle)

alarm = Timer(0.1, True)
rotator = RotationHandler(sampleCircle)
alarm.addHandler(rotator)
alarm.start( )
```

- This program rotates the circle around its center every tenth of a second. This will continue until the canvas is closed.
- **Monitors**
- The **Monitor** class supports two methods, *wait()* and *release()*. When *wait()* is called, control of that flow will not be returned until the monitor is somehow released. Here is an example where **Monitor** is used.

```
1  class ShapeHandler(EventHandler):
2    def __init__(self, monitor):
3      EventHandler.__init__(self)
4      self._monitor = monitor
5
6    def handle(self, event):
7      if event.getDescription( ) == 'mouse drag':
8        self._monitor.release( )
9
10 paper = Canvas( )
11 checkpoint = Monitor( )
12 handler = ShapeHandler(checkpoint)
13
14 cir = Circle(10, Point(50,50))
15 cir.setFillColor('blue')
16 cir.addHandler(handler)
17 paper.add(cir)
18 square = Square(20, Point(25,75))
19 square.setFillColor('red')
20 square.addHandler(handler)
21 paper.add(square)
22
23 checkpoint.wait( )
24 paper.setBackgroundColor('green')
```

FIGURE 15.9: The use of a monitor.

- **Programming Using Events**
- **Adding and Moving Shapes on a Canvas:** Here is a program. Explain what happens.

```
# Program: NewShapeHandler.py
# Authors: Michael H. Goldwasser
#          David Letscher
#
#
from cs1graphics import *

class ShapeHandler(EventHandler):
  def __init__(self):
    EventHandler.__init__(self)
    self._mouseDragged = False

  def handle(self, event):
    shape = event.getTrigger()
    if event.getDescription() == 'mouse drag':
      old = event.getOldMouseLocation()
      new = event.getMouseLocation()
      shape.move(new.getX()-old.getX(), new.getY()-old.getY())
      self._mouseDragged = True
    elif event.getDescription() == 'mouse click':
      self._mouseDragged = False
    elif event.getDescription() == 'mouse release':
      if not self._mouseDragged:
        shape.scale(1.5)
    elif event.getDescription() == 'keyboard':
      shape.setFillColor(Color.randomColor())

class NewShapeHandler(EventHandler):
  def __init__(self):
    EventHandler.__init__(self)
    self._shapeCode = 0
    self._handler = ShapeHandler()
        # single instance handles all shapes

  def handle(self, event):
    if event.getDescription() == 'mouse click':
      if self._shapeCode == 0:
        s = Circle(10)
```

```
        elif self._shapeCode == 1:
          s = Square(10)
        elif self._shapeCode == 2:
          s = Rectangle(15,5)
        elif self._shapeCode == 3:
          s = Polygon(Point(5,5), Point(0,-5), Point(-5,5))
        self._shapeCode = (self._shapeCode + 1) % 4
                # advance cyclically

        s.move(event.getMouseLocation().getX(), event.
           getMouseLocation().getY())
        s.setFillColor('white')
        event.getTrigger().add(s)
                # add shape to the underlying canvas
        s.addHandler(self._handler)
                # register the ShapeHandler with the new shape

if __name__ == '__main__':
  paper = Canvas(400, 300, 'white', 'Click me!')
  paper.addHandler(NewShapeHandler())
                # instantiate handler and register all at once
```

- **A Dialog Box Class**
- Here is another program; explain what it does.

```
# Program: Dialog.py
# Authors: Michael H. Goldwasser
#          David Letscher
#
#
from cs1graphics import *

class Dialog(EventHandler):    # Note: handles itself!
  """Provides a pop-up dialog box offering a set of choices."""

  def __init__(self, prompt='Continue?', options=('Yes', 'No'),
               title = 'User response needed', width=300, height
                  =100):
    """Create a new Dialog instance but does not yet display it.

    prompt      the displayed string (default 'Continue?')
```

```
    options     a sequence of strings, offered as options (default
      ('Yes', 'No') )
    title       string used for window title bar (default 'User
      response needed')
    width       width of the pop-up window (default 300)
    height      height of the pop-up window (default 100)
    """
    EventHandler.__init__(self)
    self._popup = Canvas(width, height, 'lightgray', title)
    self._popup.close()         # hide, for now
    self._popup.add(Text(prompt, 14, Point(width/2,20)))

    xCoord = (width - 70*(len(options)-1))/2
                                # Center buttons
    for opt in options:
      b = Button(opt, Point(xCoord, height-30))
      b.addHandler(self)  # we will handle this button ourselves
      self._popup.add(b)
      xCoord += 70

    self._monitor = Monitor()
    self._response = None

  def display(self):
    """Display the dialog, wait for a response and return the
      answer."""
    self._response = None  # clear old responses
    self._popup.open()     # make dialog visible
    self._monitor.wait()   # wait until some button is pressed
    self._popup.close()    # then close the popup window
    return self._response # and return the user's response

  def handle(self, event):
    """Check if the event was a mouse click and have the dialog
      return."""
    if event.getDescription() == 'mouse click':
      self._response = event.getTrigger().getMessage()
                          # label of chosen option
      self._monitor.release()
                          # ready to end dialog
```

```
if __name__ == '__main__':
  survey = Dialog('How would you rate this interface?',
                  ('good', 'so-so', 'poor'), 'User Survey')

  answer = survey.display()   # waits for user response
  if answer != 'good':
    print "Let's see you do better (see exercises)"
```

- **A Stopwatch Widget**
- Explain what the following program does.

```
# Program: Stopwatch.py
# Authors: Michael H. Goldwasser
#          David Letscher
#
#
from cs1graphics import *

class Stopwatch(Layer, EventHandler):
  """Display a stopwatch with start, stop, and reset buttons."""

  def __init__(self):
    """Create a new Stopwatch instance."""
    Layer.__init__(self)
    EventHandler.__init__(self)

    border = Rectangle(200,100)
    border.setFillColor('white')
    border.setDepth(52)
    self._display = Text('0:00', 36, Point(0,-20))

    self._start = Square(40, Point(-60,25))
    self._stop  = Square(40, Point(0,25))
    self._reset = Square(40, Point(60,25))
    buttons = [self._start, self._stop, self._reset]
    for b in buttons:
      b.setFillColor('lightgray')
      b.setDepth(51)  # in front of border, but behind icons
```

```
    self._startIcon = Polygon( Point(-70,15), Point(-70,35), Point
        (-50,25) )
    self._startIcon.setFillColor('black')
    self._stopIcon = Square(20, Point(0,25))
    self._stopIcon.setFillColor('black')
    self._resetIcon = Text('00', 24, Point(60,25))
    buttons.extend([self._startIcon, self._stopIcon, self.
        _resetIcon])
    for obj in buttons + [self._display, border]:
      self.add(obj)   # add to the layer

    self._clock = 0   # measured in seconds
    self._timer = Timer(1, True)
    for active in [self._timer] + buttons:
      active.addHandler(self) # we will handle all such events

  def getTime(self):
    """Convert the clock's time to a string with minutes and
        seconds."""
    min = str(self._clock // 60)
    sec = str(self._clock % 60)
    if len(sec) == 1:
      sec = '0'+sec   # pad with leading zero
    return min + ':' + sec

  def handle(self, event):
    """Deal with each of the possible events.

    The possibilities are timer events for advancing the clock,
    and mouse clicks on one of the buttons.
    """
    if event.getDescription() == 'timer':
      self._clock += 1
      self._display.setMessage(self.getTime())
    elif event.getDescription() == 'mouse click':
      if event.getTrigger() in (self._start, self._startIcon):
        self._timer.start()
      elif event.getTrigger() in (self._stop, self._stopIcon):
        self._timer.stop()
      else:  # must have been self._reset or self._resetIcon
```

```
        self._clock = 0
        self._display.setMessage(self.getTime())

if __name__ == '__main__':
    paper = Canvas(400,400)
    clock = Stopwatch()
    paper.add(clock)
    clock.move(200,200)
```

- **A Network Primer**
- Every computer has an IP (Internet Protocol) address. It is a numerical label assigned to each device connected to a computer network that uses the Internet Protocol for communication.
- In IPv4 (Internet Protocol version 4), an IP address consists of 4 numbers separated by dots. Each of these 4 numbers is a number from 0 to 255. In other words, in IPv4, an IP address takes up 4 bytes (= 32 bits).
- IPv6 is under deployment now, and it uses128 bits.
- To find your computer's Public IP address, simple google search.
- To find your computer's Private IP address, open your command prompt (windows) or terminal (mac) and type ipconfig (windows) or ifconfig (mac). Hidden among all those lines is the Private IP address for your computer.
- For a mac, there is another way of finding your computer's private IP address. Another way is to go to *System Preferences –> Network –> Wi-Fi* (or TCP/IP) to find IP number.
- The string of characters we type to go to a website is known as **URL (uniform resource locator).** For example, *http://www.prenhall.com/goldwasser*
- The substring *www.prenhall.com* is termed a **host name** and symbolizes a machine somewhere on the network.
- Behind the scene, network connections are established using a number known as an **IP address**, example, 168.146.73.101. To translate a host name to its underlying IP address, your computer will send a query to a special computer on the internet known as a **domain name server (DNS)**.
- To find the IP address of prenhall.com, go to your command prompt or terminal and type ping prenhall.com. You will find the static IP address of the host name. You will need to stop the repeated pings by CTRL-C.
- When you ping prenhall.com you will see the IP address 168.146.73.101. When you ping google.com you will see the IP address 172.217.12.174.
- Open a browser and type in these IP addresses to open the respective addresses.
- Consider the web-address

  http://www.bcc.cuny.edu/academics/academic-programs/

  The */academics/academic-programs/* portion is a path to a particular information on bcc.cuny.edu server.
- The *http* portion of the URL denotes a particular **network protocol** that will be used for the transmission of information. In this case, the **HyperText Transfer Protocol.**
- Since networks are used to transmit information, the sender and receiver must have an agreement as to the particular format that will be used.
- Because so many different protocols exist, a machine will often execute different software depending upon the data format being used.

- To help quickly segment incoming network traffic according to the protocol, each network connection to a remote machine must be designated at a specific **port**. To find port numbers, go to command-prompt or terminal and type netstat -a -b -n. As long as you have administrative authority, you will see various ports on your computer.
- A computer port is a type of electronic, software- or programming-related docking point through which information flows from a program on your computer or to your computer from the Internet or another computer in a network. A network is a series of computers that are physically or electronically linked.
- In computer terms, a computer or a program connects to somewhere or something else on the Internet via a port. Port numbers and the user's IP address combine into the "who does what" information kept by every Internet Service Provider.
- A port is a number ranging from 0 to 65535 (16 bits). There are established conventions for using certain port numbers for certain types of activity.
- For example, queries to a web server are typically sent through port 80, and so a machine might automatically handle network traffic from that port with software designated for *http* activity. For *https*, the port number is 443.
- For instance, go to your web-browser and type in google.com:80 or google.com:443. Also try google.com:20
- To find some of the common port numbers, visit
  https://www.webopedia.com/quick_ref/portnumbers.asp
- **A network socket**
- To manage the use of a specific network connection, programmers rely upon an abstraction known as a **socket.** A socket is one endpoint of a two-way communication link between two programs running on the network.
- A socket is the combination of an IP address plus port. For example, if you are looking at the Google website, then there is a connection between your computer and the Google server. Your computer's IP number+port is the socket at your end. Google's IP number and port is the socket at the other end.
- For a connection between two machines, a separate socket is maintained at each end, used to track the incoming and outgoing data for that machine.
- As a programmer, the socket serves as the interface for sending or receiving data from the opposite machine. In Python, this is modeled using a **socket** class which is imported from the **socket** module.

```
from socket import socket
s = socket() #This command creates a socket.
```

- To open a network connection to a remote host, we must specify both the host address and the network port number:

```
s.connect( ('www.prenhall.com', 80))
```

- This class supports a variety of behaviors. Some are summarized here:

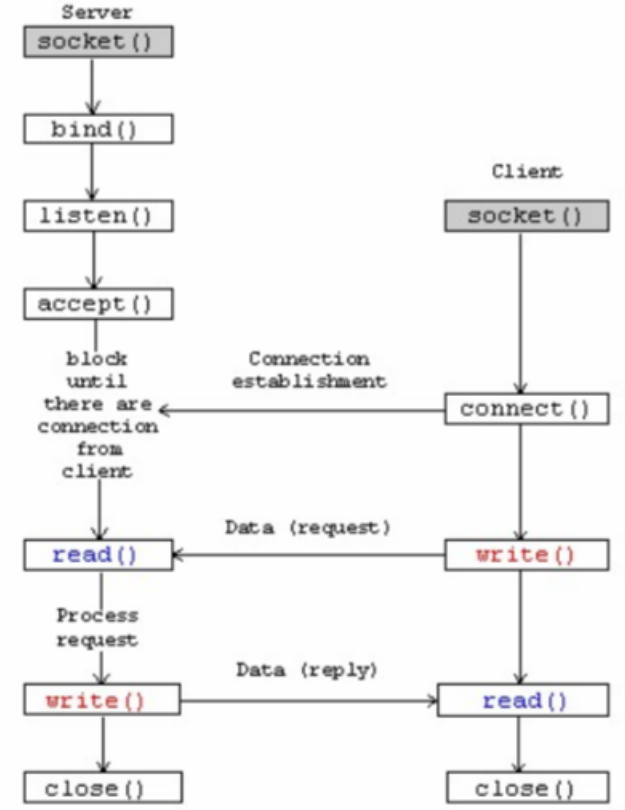
Syntax	Semantics
s.connect((host,port))	Connects socket s to the remote host at the specified port, raising an error if the connection fails.
s.send(data)	Sends the given string to the remote host.
s.recv(maxNumChars)	Returns a string of character that have been received from the remote host, up to the specified maximum number. If fewer characters are returned, this means that no further characters are currently available. If an empty string is returned, this signifies that the connection is no longer open. If the connection is open yet no new data has been received, this call waits indefinitely before returning.
s.close( )	Disconnects socket s from the remote host (this instance cannot be reconnected).

FIGURE 16.1: Selected behaviors of Python's socket class, on prototypical instance s.

- Here are some more socket commands:
- *socket.bind(host, port)* This binds the socket with the host (IP address) and the port. Note, bind() is different from connect().
- *bind()* is called assigning a name to a socket. It associates the socket with its local address. In other words, the server side binds, so that clients can use this address to connect to server.
- *connect()* is used to connect to a remote (server) address. In other words, *connect()* is on the client side.
- *socket.listen()* The server is listening whether any data has been received.
- The *send()* method is used to transmit data to the remote host. The parameter is a string to be transmitted.
- The *recv()* method is used receive data from the remote host and returns a string of characters. Unlike reading from a file, the *recv()* method requires a parameter to designate the maximum number of characters that we are willing to accept at the moment. It is not possible to accept an unbounded amount of data at one time.
- The *client* connects to the *server* to make a request for information. An analogy is, an applicant calls an employer for a job. Note, the employer does not even need to know the existence of an applicant. Likewise, the server does not need to know even the existence of a client. But the client needs to know the existence, and the address of the server.
- The client and the server have their own steps in establishing their own sockets.
- On the *client* side, the following steps are needed:
  1. Create a socket with the *socket()* method;
  2. Connect the socket to the address of the server using the *connect()* method;
  3. Send and receive data typically using *write()* and *read()* methods.
- On the *server* side, the following steps are needed:
  1. Create a socket with the *socket()* method;
  2. Bind the socket to an address using the *bind()* method.
  3. Listen for connections with the *listen()* method;

4. Accept a connection with the *accept()* method. Note, this call typically blocks until a client connects with the server.

5. Send and receive data. The *accept()* method extracts the first connection request on the queue of pending connections for the listening socket, creates a new connected socket, and returns a new file descriptor referring to that socket. Once you have that new file descriptor, you can use *recv()* to receive data from the client on it.



- When a call to *recv()* is made, there are four possible responses.
- *First response:* The call returns a string of the indicated length. There may exist additional data sent through the network, yet not returned by this call due to the specified limit. Such further data is buffered by the **socket** instance and returned by a subsequent call to *recv()*.
- *Second response:* The call returns less data than the specified maximum, presumably because no further data has been sent by the remote host.
- *Third response:* No additional data is available. Rather than return empty handed, this call waits indefinitely for more data to be received through the network. This may be problematic if the remote host has no intention of sending data.
- *Fourth response:* The call returns an empty string. This indicates that the connection has been closed either by the remote host or a network disruption, so there is no reason to continue waiting.
- We can terminate a network connection from our end by invoking the *close()* method upon the socket. Once we have closed a connection, any subsequent call to *send()* or *recv()* will result in an error.

- **Writing a Basic Client**
- The two most common models for network communication are: **client-server** model and **peer-to-peer (P2P)** model.
- In a client-server model the machine acting as a **server** waits for a **client** to initiate contact. For example, a web server waits for someone to make an explicit request to receive content from that website.
- The protocols used for the client-server module involves the client sending a request and the server sending a response to that request. The software for controlling a client is usually different from the software for controlling a server.
- In a peer-to-peer network, two connected machines have more symmetric capabilities and thus each run the same software.
- Peer-to-peer software is often implemented by combining the separate capabilities seen in client software and server software.
- **Fetching the Time and Date**
- The National Institute of Science and Technology (NIST) runs servers that, when queried, report the current time.
- These servers are connected to atomic clocks so their time is precise (although the time when it is received may not be precise due to network delay).
- Time servers support a simple protocol on port 13 known as **daytime**.
- Whenever a client opens a connection to a time server, the server immediately returns a string that contains the time and date information. The client need not even send an explicit request; the mere act of opening a connection implies the client's desire to know the time.
- Try the following:

```
from socket import socket  #importing a the socket class
connection = socket() #instantiate an instance of the socket class
connection.connect(('time.nist.gov', 13))
        #establish a connection to one of NIST's servers
print(connection.recv(1024))
        #the server sends a formatted string which
        #we retrieve and print;
        #the value 1024 is the maximum number of bytes
        #of data we are willing to receive
```

- You should obtain a string which resembles '\n58500 19-01-17 11:35:00 00 0 0 710.3 UTC(NIST) * \n'
- Here, UTC stands for Universal Time Coordinated (formerly known GMT – Greenwich Mean Time).
- Here is another way of printing:

```
from socket import socket
connection = socket()
connection.connect(('time.nist.gov', 13))
```

```
fields = connection.recv(1024).split()
date = fields[1].decode()
time = fields[2].decode()
print('Date (YY-MM-DD) is %s, and time is %s (UTC)' %(date, time))
```

- **Downloading a Web Page**
- We now develop a client that downloads and saves a web page. After the connection is made, the client must explicitly send a request to the server for the desired web page.
- Note that data received (or sent) is in 'UTF-8' code. For efficient storage of strings, the sequence of unicode points are stored into set of bytes. This is encoding. For example, try the following on your shell:

```
a = 'This is a string. We want to encode it.'
a.enclode('utf-8') #see the output.
a.encode('utf-16') #now see the output.
a.encode('utf-32') #what does this look like?
```

- By default, the encode() method returns 'utf-8' encoded version of the string.
- For more on encoding visit
  https://docs.python.org/3/library/codecs.html#standard-encodings
- Likewise, when we receive data, it is encoded. Therefore, we must decode that data.
- Here is a complete program to download a single web page and save the contents (but not the header) to a file. Try the program with the website
  'http://fsw01.bcc.cuny.edu/mathdepartment/Courses/Math/MTH23/math23.htm'

```
# Program: webclient.py
# Authors: Michael H. Goldwasser
#          David Letscher
#
# This example is discussed in Chapter 16 of the book
# Object-Oriented Programming in Python
#
from socket import socket

        #Try: http://www.bcc.cuny.edu/academics/academic-programs/
print('Enter the web page you want to download.')
print('Use the format http://domain.name/page/to/download')

url = input()
server = url.split('/')[2]                    # e.g., domain.name
page = '/' + '/'.join(url.split('/')[3:])     # e.g., /page/to/
   download
connection = socket()
```

```
connection.connect( (server, 80) )
connection.send(str.encode('GET %s HTTP/1.0\r\n\r\n' %page))
          #\r stands for CR (Carriage Return)
          #\n stands for LF (Line Feed)
  #The request 'GET /index.html HTTP/1.0' asks the
  #server to send the index.html page using version 1.0 of HTTP
  #protocol.
raw = connection.recv(1024).decode('utf-8')
      # read first block (header + some content)

sep = raw.index('\r\n\r\n')
header = raw[ :sep]
content = raw[sep+4: ]
    #Each line of th header is terminated with '\r\n' and
    #the entire header will not have any blank lines. But the
       header
    #ends with a blank line. So, \r\n\r\n captures this end of the
        header.



outputFile = open('download.html', 'w')
outputFile.write(content)
      # write out content we have seen thus far


while len(content) > 0:    # still more possible content...
  content = connection.recv(1024).decode('utf-8')
  outputFile.write(content) #keep writing to the outputFile


      #Looks like no content is left (that is, len(content) = 0)
outputFile.close()
connection.close()
```

- Our program for downloading a webpage shows how low-level steps are executed when retrieving a webpage from a server using the HTTP protocol.
- Note, this task is high level, and fairly common in practice. So, Python has a library (*urllib*) which contains several tools for web processing.
- In particular, there is a function *urlopen* which takes a string parameter specifying a URL and returns a file-like object from which we can read the downloaded contents. Moreover, this function uses a more modern version of HTTP than 1.0 (presently on HTTP 3.0).

- Here is a high-level way of downloading a webpage:

```
from urllib import request
print('Enter the webpage you want to download')
print('Use the format http://domain.name/page/')
print('For example, http://www.bcc.cuny.edu/academics/academic-
    programs/')
url = input()

content = request.urlopen(url)
contentString = content.read()

outputFie = open('download.html', 'w')
print(contentString, file = outputFile)
outputFile.close()
```

- Writing a server is quite different from writing a client.
- A client typically establishes a connection, makes a request, and processes the response.
- Implementing a server at a low level consists of listening on a particular port for incoming connections, creating a socket to manage each such connection, and then following the chosen network protocol for communication with the client.
- Python provides a **socketserver** module with some convenient tools to help write a server.
- **TCP** stands for **Transmission Control Protocol** used for establishing connections on the internet.
- The **TCPServer** class handles all the details of listening on a specified port for incoming connections and creating a dedicated socket for each such connection.
- The precise interactions that should take place once a client has connected to the server needs to be customized.
- For this, we provide a second class that will be used to handle each connection (event-driven programming).
- We define a special-purpose class that inherits from the **BaseRequestHandler** class (also imported from **socketserver** module).
- Every time a new connection is established by a client, a **handle** method is called.
- By the time the body of the **handle** method is executed, the **TCPServer** will have already initialized a socket, identified as *self.request*, that can be used for communicating with the client.
- Note that in this case, the the server's socket uses the *send* method to send data from the server to the client. Likewise, the *recv* method is used to retrieve data from the client to the server.
- The port can be whatever number we choose (other than those reserved for special protocols). But the client needs to reach out to the same port.
- The **HandlerClass** is the name of the customized handler class, as opposed to an instance of that class.
- The method *serve_forever()* tells the server to continue running so long as the Python program is executing. We can terminate the server by interrupting the Python interpreter.
- Here is the echo server:

```
# Program: echoserver.py
# Authors: Michael H. Goldwasser
#          David Letscher
#

from socketserver import TCPServer, BaseRequestHandler

class EchoHandler(BaseRequestHandler):
  def handle(self):
    '''This overrides the handle method of the parent class.'''
```

```
        message = self.request.recv(1024)
            #self.request references the controlling socket.
        self.request.send(message) #This is the echo


    # may need to customize localhost and port for your machine
    echoServer = TCPServer( ('localhost', 9000), EchoHandler)
          #Ther Server instance.
    echoServer.serve_forever()
          #This activates the server
```

- Once the server is up and running, test it. Start a second Python interpreter session on the same machine (or even some other machine) and try the following:

```
from socket import socket
echo = socket()
echo.connect( ('localhost' , 9000)) #if you are on the same
   machine. Else, find the IP of the machine with the server.
echo.send("This is a test'.encode())
print(echo.recv(1024).decode())
```

- The return value is the number of characters that were successfully sent. This is provided by the socket (not the echo server).
- If we try to echo a second message on this existing socket, we will find that the connection has been closed.
- But as long as the echo server is executing, we could create a new socket and echo.
- **Basic Web Server**
- A web server is used to allow others access to content stored on the server's machine.
- A web server is a computer system that processes requests via HTTP, the basic network protocol used to distribute information on the World Wide Web.
- We present a code for a web server below.Note, our intent is to only give access to files or directories located in the same place that we run our script. If we are unable to open the file, we report this error using a standard HTTP protocol.
- Note, instead of the standard port number 80, we use the port number 8080.
- **Word of Warning:**
- Running our web server implementation on your computer for any extended length of time is a security risk. Do not leave it running after trying it out.

```
# Program: webserver.py
# Authors: Michael H. Goldwasser
#          David Letscher
#
from socketserver import TCPServer, BaseRequestHandler


class WebHandler(BaseRequestHandler):
```

```
    def handle(self):
        '''Overriding the handle method of the base class.'''

        command = self.request.recv(1024).decode('utf-8')

        if command[:3] == 'GET':
            pagename = command.split()[1][1:]
                #Identify the name of the file, and
                # remove leading '/' for filename

            try:
                requestedFile = open(pagename, 'r')
                content = requestedFile.read()
                requestedFile.close()
                header = 'HTTP/1.0 200 OK\r\n'
                    #standard header
                self.request.send(header.encode('utf-8'))
                self.request.send(content.encode('utf-8'))

            except IOError: # could not open the file
                self.request.send('HTTP/1.0 404 Not Found\r\n\r\n'.encode
                    ())

    webServer = TCPServer( ('localhost', 8080), WebHandler)
    webServer.serve_forever()
```

- On a separate interpreter, type the following:

```
from socket import socket
s = socket()
s.connect(('localhost', 8080))
fileName = 'trial.ext'
message = 'GET /'+fileName
s.send(message.encode())
print(s.recv(1024).decode('utf-8'))
```

- Note, if the file you asked for is larger than the limit of 1024 bytes, then you would have to print the next chunks again and again.

- **Network Chat Room**
- We want to build a client-server model for a network chat room.
- Unlike the previous case, we need to specify our protocols. For instance, unlike the echo server, here when a client connects to the server, they may stay connected for quite a while. This is called a **persistent connection.**
- In a persistent connection, the server cannot be solely devoted to handling that connection until it is closed, or else it will not be able to respond to other clients trying to connect.
- Here is a persistent echo server:

```
# Program: persistentEchoServer.py
#

from socketserver import TCPServer, BaseRequestHandler

class EchoHandler(BaseRequestHandler):
  def handle(self):
    '''This overrides the handle method of the parent class.'''
    active = True
    while active:

      message = self.request.recv(1024).decode()
          #self.request references the controlling socket.
      if message.lower() == 'quit':
        self.request.send(("ECHO: Goodbye. ").encode())
        active = False #Breaks the while loop
      else:
        self.request.send(("ECHO: " + message).encode()) #This is
           the echo



    self.request.close() #closes our socket at the end of the
       while loop



# may need to customize localhost and port for your machine
echoServer = TCPServer( ('localhost', 9000), EchoHandler)
      #Ther Server instance.
echoServer.serve_forever()
      #This activates the server

'''
```

```
On the client side, try the following on an interpreter:

from socket import socket
echo = socket()
echo.connect(('localhost',9000))
echo.send('this is a test.'.encode())
print(echo.recv(1024).decode())
echo.send('hello.'.encode())
print(echo.recv(1024).decode())
echo.send('quit'.encode())
print(echo.recv(1024).decode())
echo.send('hello.'.encode())
print(echo.recv(1024).decode())

'''
```

- Likewise, the client also needs to do two things: one, monitoring the keyboard in case the user enters any commands, and two, monitoring the socket to listen for transmissions from the server.
- So we will use **multithreading.** This allows several different pieces of software to execute at the same time.
- Try the following:

```
from threading import Thread
from time import sleep

class MyThread(Thread):
    def run(self): #overrides the run() method of Thread class.
        print('Second thread begins.')
        sleep(5)
        print('Second thread is done.')
secondThread = MyThread()
secondThread.start()  #indirectly calls the run() method.
print('The program is done.')
```

- Here is a flow of our multithreaded programme.

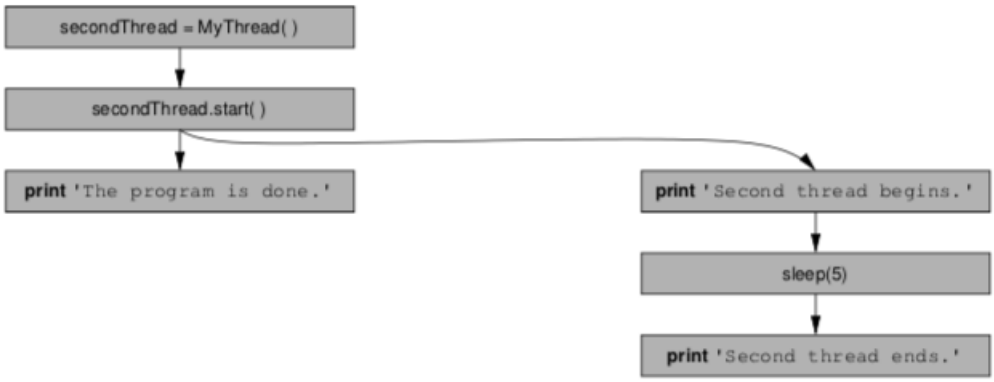


FIGURE 16.12: The flow of a multithreaded program.

- On some systems, the first two of the output statements may be reversed as the two threads compete with each other for time on the processor.
- **Communication Protocol:**
- We want to differentiate between messages that users send to each other and messages that the client and server software send to each other.
- Users should be allowed to come and go or remain.
- Some messages are meant for everyone in the room, and some are private.
- The user interacts with the client software.
- There are four events for a particular user at the client side: the user may join the room, the user may quit the room, the user may send a message to the entire room, or the user may send a message to a specified individual.

Message Type	Format
Join the room using given identity	`'ADD %s\n' % screenName`
Broadcast the message to everyone	`'MESSAGE %s\n' % content`
Send a private message	`'PRIVATE %s\n%s\n' % (recipient,content)`
Quit the chat room	`'QUIT\n'`

FIGURE 16.8: The protocol for messages from the client to the server.

- The server will inform a client whenever another individual joins or leaves the room, and whenever messages have been entered that are supposed to be seen by this client's particular user.
- We differentiate between messages that are viewable because they were broadcast to the whole room and messages that are viewable because they were privately sent to this user.
- Finally, the server will explicitly acknowledge when it receives word that a particular user is quitting the chat room.

Message Type	Format
New user has joined	`'NEW %s\n' % screenName`
Message was broadcast to everyone	`'MESSAGE %s\n%s\n' % (sender, content)`
Private message was sent to user	`'PRIVATE %s\n%s\n' % (sender, content)`
Someone has left the room	`'LEFT %s\n' % screenName`
Acknowledges request to quit	`'GOODBYE\n'`

FIGURE 16.9: The protocol for messages from the server to the client.

- **The Server**
- Most of the work involves the definition of a **ChatHandler** class that specializes **BaseRequestHandler** (similar to what we did for our echo server and our web server).
- The primary difference is as follows: For the echo server and the web server, each connection represented a single query, which is processed and dismissed. In the chat server, the **handle** method is called whenever someone connects to the server and the routine continues for the full duration of that user's participation.
- That is, the **handle** routine consists of one big while loop that begins as soon as a client connects. It remains active until finally receiving the QUIT command from the client or until the connection fails (when *recv* returns an empty transmission).
- The *_socketLookup* dictionary is maintained to map a screen name to the actual socket that is being used to manage the connection to that user's client.
- A new entry is added to the dictionary when a user initially joins the room, and the user's socket removed after the connection is closed – the socket **self.request** is closed and the *username* key-value pair is popped from the dictionary.
- The **_broadcast** function is a utility for transmitting network activity to each current client.
- There is a separate **ChatHandler** instance to manage each user's connection to our server.
- The **handle** method will be running for a long time as one particular user remains in the chat room.
- An important distinction between this implementation and our earlier servers is that we use the **ThreadingTCPServer** class, not the **TCPServer** class. The threaded version of the TCP server uses multithreading.
- Here is the *ChatHandler*:

```
from socketserver import ThreadingTCPServer, BaseRequestHandler

# A global dictionary.
# This associates usernames to sockets.
_socketLookup = dict()

# Send the announcement to all users.
def _broadcast(announcement):
    for connection in _socketLookup.values():
        connection.send(announcement.encode())
```

```
class ChatHandler(BaseRequestHandler):
    def handle(self): #Overriding the handle method of the base
        class
         # We will change this as needed.
         username = 'Unknown'
         active = True
         while active:
             # Wait for something to happen.
             transmission = self.request.recv(1024).decode()

             if transmission:
                 # The transmission should have the form '[COMMAND]
                     [DATA]'
                 command = transmission.split()[0]
                 data = transmission[1+len(command):]

                 if command == 'ADD':
                     username = data.strip() #removes leading and
                         trailing whitespace
                     _socketLookup[username] = self.request #
                         building the dictionary
                     _broadcast('NEW %s\n'%username)
                 elif command == 'MESSAGE':
                     _broadcast('MESSAGE %s\n%s\n'%(username, data)
                         )
                 elif command == 'PRIVATE':
                     recipient = data.split('\n')[0]
                     if recipient in _socketLookup:
                         content = data.split('\n')[1]
                         _socketLookup[recipient].send(('PRIVATE %s
                             \n%s\n'%(username, content)).encode())
                 elif command == 'QUIT':
                     active = False
                     # Tell the user we saw the 'QUIT'.
                     self.request.send('GOODBYE\n'.encode()) #
                         acknowledge
             else:
                 # If the transmission variable is not set, then
                     the socket failed.
```

```
                active = False  #socket has failed

            # The ChatHandler is no longer active.
            # Either the user QUIT or the socket failed.
            self.request.close()
            _socketLookup.pop(username)
            _broadcast('LEFT %s\n' % username)

# Create a Threading TCP Server.
# It should listen on this machine (localhost) at port 9000.
# Each incoming connection is handled by a new ChatHandler.
myServer = ThreadingTCPServer( ('localhost', 9000), ChatHandler)

# Start this thing up ...
myServer.serve_forever()
```

- **The Client**
- The client needs multithreading for doing two things simultaneously.
- One is that we want the user to type commands, while the other that the user should be able to see messages from other users.
- Using *input()* will block our flow of control. Likewise, only using *recv()* (which is meant to listen for activity on the network socket) will block our flow of control.
- Two threads share a single network socket identified as *server*.
- The **IncomingThread** class is devoted to monitoring incoming messages from that socket. It executes a while loop that repeatedly waits for more information from the server. When receiving network traffic, it uses knowledge of our protocol to interpret the meaning. The symbol ==> helps demarcate the session.
- The primary thread begins by establishing the shared socket connection and registering the user in the chat room using the ADD protocol. It is from the primary thread that the secondary thread for monitoring incoming network activity is spawned.
- The remaining responsibility of the primary thread is to monitor input entered by the local user via the keyboard, while the secondary thread continues to run independently.
- Note, the *network protocol* uses capitalized words, but the user is not required to.
- Here is the code for the *ChatClient*:

```
# Program: chatclient.py
# Authors: Michael H. Goldwasser
#          David Letscher
#
# This example is discussed in Chapter 16 of the book
# Object-Oriented Programming in Python
#
from socket import socket
from threading import Thread

class IncomingThread(Thread):
  def run(self):
    stillChatting = True
    while stillChatting:                    # wait for more
       incoming data
      transmission = server.recv(1024).decode()        # 'server'
         will be defined globally at line 27
      lines = transmission.split('\n')[:-1]
      i = 0
      while i < len(lines):
        command = lines[i].split()[0]          # first keyword
        param = lines[i][len(command)+1: ]     # remaining
           information
```

```
            if command == 'GOODBYE':
              stillChatting = False
            elif command == 'NEW':
              print('==> %s has joined the chat room' % param)
            elif command == 'LEFT':
              print('==> %s has left the chat room' % param)
            elif command == 'MESSAGE':
              i += 1                                  # need next line for
                 content
              print('==> %s: %s' % (param, lines[i]))
            elif command == 'PRIVATE':
              i += 1                                  # need next line for
                 content
              print('==> %s [private]: %s' % (param, lines[i]))
            i += 1

instructions = """
-------------------------------------------------------
Welcome to the chat room.

To quit, use syntax,
  quit

To send private message to 'Joe' use syntax,
  private Joe:how are you?

To send message to everyone, use syntax,
  hi everyone!
-------------------------------------------------------
"""

server = socket()                                     # shared by
   both threads
server.connect( ('localhost', 9000) )                 # could be a
   remote host
username = input('What is your name: ').strip()
server.send(('ADD %s\n' % username).encode() )
incoming = IncomingThread()
incoming.start()
```

```
print(instructions)
active = True                                    # main thread
    for user input
while active:
    message = input()                            # wait for more
        user input
    if message.strip():
        if message.rstrip().lower() == 'quit':
            server.send('QUIT\n'.encode())
            active = False
        elif message.split()[0].lower() == 'private':
            colon = message.index(':')
            friend = message[7:colon].strip()
            server.send(('PRIVATE %s\n%s\n' % (friend,message[1+colon:
                ])).encode() )
        else:
            server.send(('MESSAGE ' + message).encode())
```

- **End of the semester does not mean end of learning**
- Go through the website of Carnegie Mellon's website for their introductory course in Python.
  https://www.cs.cmu.edu/∼112/schedule.html
- Go through the following website: https://introcs.cs.princeton.edu/python/home/
- Here is Google's Python class: https://developers.google.com/edu/python/
- A good interactive book: http://interactivepython.org/courselib/static/thinkcspy/index.html
- Another useful book: https://books.trinket.io/pfe/
- There are several excellent websites which train applicants for interviews for industry.
  http://www.hackerrank.com
  http://www.leetcode.com

- TK stands for "Tool kit".
- The **TKinter** module (short for "TK interface") is the standard Python interface to the TK GUI toolkit from Scriptics (formerly developed by Sun Labs).
- First step is to create a window and a canvas inside that window.
- To create a window, try the following:

```
#Closely follows
#https://www.cs.cmu.edu/~112/index.html
#Some TKinter codes

##################################################
#createWindow.py
from tkinter import *

root = Tk() #Creates a window



##################################################
#helloWorld1.py

from tkinter import *

root = Tk()

w = Label(root, text = "Hello, World!")
            #Creates a Label widget
            #A Label widget can display text
            #or other images.
w.pack() #Tells the window to pack
            #the widget within it.
root.mainloop()
            #Without this we cannot draw into
            #the canvas. This will also
            #turn on events (keyboards etc.)
            #Application window willnot appear
            #before you enter the main loop.
print("Bye.")

##################################################
```

```
#canvas1.py

from tkinter import *

def draw(canvas, width, height):
    pass # replace with your drawing code!

def runDrawing(width=300, height=300):
    root = Tk()
    canvas = Canvas(root, width=width, height=height)
    canvas.pack()#Tells the window to pack
                 #the canvas within it.
    draw(canvas, width, height)
    root.mainloop()
    print("Bye!")

runDrawing(500, 500)

##########################################################
#hello2.py

from tkinter import *

def draw(canvas, width, height):
    canvas.create_text(200, 100, text="Hello, World!",
                       fill="purple", font="Helvetica 26 bold
                          underline")

def runDrawing(width=300, height=300):
    root = Tk()
    canvas = Canvas(root, width=width, height=height)
    canvas.pack()
    draw(canvas, width, height)
    root.mainloop()
    print("Bye!")

runDrawing(500, 500)

##########################################################
#moreCanvas1.py
```

```
def draw(canvas, width, height):
    canvas.create_text(200, 100, text="Hello, World!",
                       fill="purple", font="Helvetica 26 bold
                          underline")
    canvas.create_text(200, 100, text="It is a beautiful day.",
       anchor = SW,
                       fill="purple", font="Helvetica 26 bold
                          underline")
    #Try various anchors: SE, NE, NW, N, E, W, S
    canvas.create_rectangle(10,20, 150,200, fill = 'green')


####################################################
#moreCanvas2.py


def draw(canvas, width, height):
    canvas.create_rectangle(10,20, 150,200, fill = 'green')
    canvas.create_rectangle(40,70, 190,250, fill = 'orange')

####################################################
#moreCanvas3.py

def draw(canvas, width, height):
    canvas.create_rectangle(40,70, 190,250, fill = 'orange')
    canvas.create_rectangle(10,20, 150,200, fill = 'green')

####################################################
#moreCanvas4.py

def draw(canvas, width, height):
    margin = 10
    canvas.create_rectangle(margin,margin, width-margin,height-
       margin, fill = 'orange', width=5)
    canvas.create_rectangle(10,20, 150,200, fill = 'green',width
       =5)
          #How would you change the green rectangle to fit it in
              the center?
```

```
##########################################################
#RGB string (integer values allowed from 0 to 255)

"#%02x%02x%02x" % (255, 0,0) #Red color
"#%02x%02x%02x" % (0,255,0) #Green color
"#%02x%02x%02x" % (0,0,255) #Blue color
"#%02x%02x%02x" % (125,49,32) #Some color
#Look for more RGB combinations to find colors
##########################################################
#moreCanvas5.py

def draw(canvas, width, height):
    margin = 10
    canvas.create_rectangle(margin,margin, width-margin,height-
        margin, fill = 'orange', width=5)
    canvas.create_rectangle(10,20, 150,200, fill = 'green',width
        =5)
    canvas.create_oval(10,20,150,200,fill = 'pink')
    #Or also use tuples for points:
    #canvas.create_oval((10,20),(150,200),fill = 'pink')
    #How does one draw a circle then?

##########################################################
#moreCanvas6.py

def draw(canvas, width, height):
    margin = 10
    canvas.create_rectangle(margin,margin, width-margin,height-
        margin, fill = 'orange', width=5)
    canvas.create_rectangle(10,20, 150,200, fill = 'green',width
        =5)
    canvas.create_oval((10,20),(150,200),fill = 'pink')
    canvas.create_line(100, 50, 300, 150, fill="red", width=5)
                        #This is a line from (100,50) to (300,150)
    canvas.create_polygon(100,50,150,80,300,30,200,10, fill="
        yellow")
                        #This is a polygon joining (100,50),
                            (150,80),(300,30),
                        #(200,10) and back to (100,50).
```

```
    #We could also use a list of tuples to draw our polygon as
    #canvas.create_polygon([(100,50),(150,80),(300,30),(200,10)],
        fill="yellow")
    #Why do you think a list of points is convenient? Think append
        or pop.

##########################################################
#Drawing a clock (CMU—Fundamentals of Programming)
#Understand what each line does

import math

def draw(canvas, width, height):
    (cx, cy, r) = (width/2, height/2, min(width, height)/3)
    canvas.create_oval(cx-r, cy-r, cx+r, cy+r, fill="yellow")
    r *= 0.85 # make smaller so time labels lie inside clock face
    for hour in range(12):
        hourAngle = math.pi/2 - (2*math.pi)*(hour/12)
        hourX = cx + r * math.cos(hourAngle)
        hourY = cy - r * math.sin(hourAngle)
        label = str(hour if (hour > 0) else 12)
        canvas.create_text(hourX, hourY, text=label, font="Arial
            16 bold")

##########################################################
#moreCanvas7.py

import math
def draw(canvas, width, height):
    (cx, cy, r) = (width/2, height/2, min(width, height)/3)
    canvas.create_oval(cx-r, cy-r, cx+r, cy+r, fill="yellow")
    r *= 0.85 # make smaller so time labels lie inside clock face
    points =[]
    for hour in range(12):
        hourAngle = math.pi/2 - (2*math.pi)*(hour/12)
        hourX = cx + r * math.cos(hourAngle)
        hourY = cy - r * math.sin(hourAngle)
        points.append((hourX, hourY))
    canvas.create_polygon(points, fill = 'violet')
```

```
#############################################################
#moreCanvas8.py
#To move canvas objects

from tkinter import *

def draw(canvas, width, height):
    rect1 = canvas.create_rectangle(200,300, 280, 380, fill = '
       orange', width=5)
    rect2 = canvas.create_rectangle(10,20, 150,200, fill = 'green'
       ,width=5)
    oval1 = canvas.create_oval(10,20,150,200,fill = 'pink')

    canvas.move(rect1, 100,10)
    canvas.move(oval1,100,10)


#############################################################
#moreCanvas9.py
#Use move and sleep in combination

from tkinter import *
from time import sleep

def draw(canvas, width, height):
    oval1 = canvas.create_oval(20,0,40,20 ,fill = 'pink')
    timeDelay = .25
    for i in range(48):
        sleep(timeDelay)
        canvas.move(oval1, 0,10)
        canvas.update()

#############################################################
#moreCanvas10.py
#Ticking clock

from tkinter import *
from time import sleep
import math
```

```
def draw(canvas, width, height):
    (cx, cy, r) = (width/2, height/2, min(width, height)/3)
    canvas.create_oval(cx-r, cy-r, cx+r, cy+r, fill="yellow")
    r *= 0.85 # make smaller so time labels lie inside clock face
    points =[]
    for hour in range(12):
        hourAngle = math.pi/2 - (2*math.pi)*(hour/12)
        hourX = cx + r * math.cos(hourAngle)
        hourY = cy - r * math.sin(hourAngle)
        label = str(hour if (hour > 0) else 12)
        canvas.create_text(hourX, hourY, text=label, font="Arial
            16 bold")
        points.append((hourX, hourY))

    tick = canvas.create_line((cx,cy),(cx,cy-r),width=3)#The
        needle in the clock

    timeDelay=0.25
    for p in points:
        sleep(timeDelay)
        canvas.coords(tick,cx,cy,p[0],p[1]) #Changes coordinates
            of tick
        canvas.update()

    sleep(timeDelay)
    canvas.coords(tick, cx,cy,cx,cy-r) #See what happens if you
        don't have this



###############

#More Tkinter codes.
#Closely follows: https://effbot.org/tkinterbook
#Also check out:
#https://www.python-course.eu/python_tkinter.php
#For CSI32
```

```
from tkinter import *
#You could also say,
#import tkinter
#and use tkinter.method to work with.
#Or,
#import tkinter as t
#and then use t.method to work with.

############################################################
#Code1

from tkinter import *
def say_hi():
    print('hi there everyone')

root = Tk()
frame = Frame(root)  #We have created a frame widget.
                     #A frame is a simple container.
frame.pack()        #We pack to make the frame visible
                     #Note, the pack() method returns None.
    #So, do not do the following: Frame(root).pack() since this
    #would not give you a reference to the Frame you have created.
button = Button(frame, text="Hello World!", fg = 'red', command=
   say_hi)
                #We create a button widget as a child to the frame
                    .
                #It is labelled "Hello World!"
                #fg stands for foreground (here it is red).
                #The button takes
button.pack(side=LEFT)
                #Now, the button has been packed.
                #You could have simply said, button.pack()
                # in this case.

root.mainloop()

#A frame is a rectangular region on the screen.
```

```
#Frame widgets are used to group other widgets into complex
    layouts.


############################################################
#Code2
def say_hi():
    print('hi there everyone')

root = Tk()
frame = Frame(root)
frame.pack()
button1 = Button(frame, text="Hello World!", fg = 'red', command=
    say_hi)
button1.pack()
                #Try various packing positions button1.pack()
                # button1.pack(side = ....) side options are
                #top (default), bottom, right, left
button2 = Button(frame, text='Quit', fg = 'blue', command=frame.
    quit)
                #Another button created. Notice its command.
button2.pack() #Try various packing positions button.pack()
                # button2.pack(side = ....)
root.mainloop()
root.destroy()
            #Some environments require this command to destroy the
                window
            #Try without the destroy command to see what happens.

############################################################
#Code3
#Encapsulation approach
#

from tkinter import *

def say_hi():
    print('hi there everyone')

class App:
```

```
    def __init__(self, master):

        frame = Frame(master)
        frame.pack()

        self.button1 = Button(
            frame, text="QUIT", fg="red", command=frame.quit)

        self.button1.pack(side=LEFT)

        self.button2 = Button(frame, text="Hello, World", command=
            self.say_hi)
        self.button2.pack(side=LEFT)

    def say_hi(self):
        print("hi there, everyone!")

root = Tk()

app = App(root)

root.mainloop()
root.destroy()

#The Button widget is a standard widget used to implement various
#kinds of buttons (text, image) and you can associate a
#function or method with a button.


############################################################
#Here is how to we can specify size
#Try various combinations of commenting below.

from tkinter import *

root = Tk()
#frame = Frame(root, height=200,width=150)
frame = Frame(root)
#frame.pack_propagate(0) #don't shrink
```

```
frame.pack()

button1 = Button(frame, text="Welcome to my home.How are you?")
#button1 = Button(frame, text="Welcome to my home.",height=2,
    width=1)
#button1.pack(fill = BOTH, expand =1)
button1.pack()

root.mainloop()

#################################################################
#Yet another way of using Label widget
#Make sure that lotus.gif is in the folder you run this code,
#or use a path to reach the folder.

from tkinter import *

root = Tk()
lotus = PhotoImage(file="lotus.gif")

label = Label(root, image = lotus)
label.pack()

###########
flower = '''Nelumbo nucifera, also known as Indian lotus,
        sacred lotus, bean of India, Egyptian bean or simply lotus
            ,
        is one of two extant species of aquatic plant in the
            family
        Nelumbonaceae. It is often colloquially called a water
            lily.'''

label2 = Label(root, text=flower)
label2.pack(side="bottom")
###########
def say_like():
    print("I like lotus.")

like = Button(root, text="Like", command=say_like)
like.pack(side="left")
```

```
leave = Button(root, text="Quit", command=root.destroy)
leave.pack(side="right")

root.mainloop()

#The Label widget is used to display a text or image.



##################################################################
#Another example

from tkinter import *

root = Tk()
lotus = PhotoImage(file="lotus.gif")

label = Label(root, image = lotus)
label.pack()


def say_like():
    print("I like lotus.")

like = Button(root, text="Like", command=say_like)
like.pack(side="left")

leave = Button(root, text="Quit", command=root.destroy)
leave.pack(side="right")

canvas = Canvas(root, width = 200, height = 200)
canvas.pack(side='bottom')

canvas.create_rectangle(30,50,100,100, fill = 'yellow')

##########
flower = '''Nelumbo nucifera, also known as Indian lotus,
        sacred lotus, bean of India, Egyptian bean or simply lotus
           ,
```

```
        is one of two extant species of aquatic plant in the
            family
        Nelumbonaceae. It is often colloquially called a water
            lily.'''

label2 = Label(root, text=flower)
label2.pack(side="bottom")
############

root.mainloop()




#########################################################################
#The Message widget

from tkinter import *

root = Tk()

simon_garfunkel = '''I'd rather be a sparrow than a snail,\n
Yes I would, if I could, I surely would.'''
song = Message(root, text = simon_garfunkel)
song.config(bg='lightblue', font = ('Arial',50,'italic'))
song.pack()

root.mainloop()

#This widget is used for short text messages.
#Unlike Label, here the font can be changed.
#Message can be multilines.

#########################################################################
#The Radio Button widget

from tkinter import *

root = Tk()
```

```
v=IntVar()

question= '''Who is the author of "War and Peace"?'''
qLabel = Label(root,text=question,justify='left', padx=50)
qLabel.pack()

choice1=Radiobutton(root, text="Dickens", padx=50, variable=v,
    value=1)
choice1.pack(anchor=W)

choice2=Radiobutton(root, text="Frost", padx=50, variable=v, value
    =2)
choice2.pack(anchor=W)

choice3=Radiobutton(root, text="Tolstoy", padx=50, variable=v,
    value=3)
choice3.pack(anchor=W)

def tellAnswer():
    print(v.get())
yourAnswer = Button(root, text="Your Answer", command=tellAnswer)
yourAnswer.pack(side="bottom")

root.mainloop()

#Radio button allows the user to choose exactly one of a
#predefined set of options.

##########################################################################
#Entry widget and Grid geometry

from tkinter import *


def show_entry_fields():
   print("First: %s\nLast: %s" % (e1.get(), e2.get()))

master = Tk()
first = Label(master, text="First")
```

```
first.grid(row=0)  #notice, we are not packing.
                   #we are placing first in a grid
second = Label(master, text="Second")
second.grid(row=1)

e1 = Entry(master) #This is an entry widget
e2 = Entry(master)

e1.grid(row=0, column=1)  #grid again...
e2.grid(row=1, column=1)

qbutton = Button(master, text='Quit', command=master.quit)
#qbutton.grid(row=3, column=0)
qbutton.grid(row=3, column=0, sticky=W, pady=20)
sbutton = Button(master, text='Show', command=show_entry_fields)
#sbutton.grid(row=3, column=1)
sbutton.grid(row=3, column=1, sticky=W,pady=20)



#lotus = PhotoImage(file="lotus.gif")
#label = Label(master, image = lotus)
#label.grid(row=0, column=2, columnspan=2, rowspan=2)

master.mainloop( )
master.destroy()

#Entry widgets allow us to get input from the user in
#the form of a text string. The content gets scrolled if
#the available display space is not enough.

#The Grid allows us to put widgets in a 2-dimensional table.




##################################################################
#Events and Binds

from tkinter import *

root = Tk()
```

```
def giveChar(event):
    print("You pressed", repr(event.char))


def giveCoords(event):
    frame.focus_set()
        #Keyboard events require focus to be
        #sent to the widget, even if you
        #do not keep track of the coordinates.
    print("You clicked at", event.x, event.y)


frame = Frame(root, width=200, height=200)
frame.bind("<Key>", giveChar)
    #<Key> is a specific event.
frame.bind("<Button-1>",giveCoords)
    #<Button-1> is a specific event.
frame.pack()


root.mainloop()


#Python provides a mechanism to bind functions and methods to
   widgets.
#The syntax is widget.bind(event, handler)
#To learn more, see
#https://www.python-course.eu/tkinter_events_binds.php
#Here is binding with a Canvas object.
##########################################################################
##########################################################################

import tkinter as tk

def giveCoords(event):
    canvas.focus_set()
        #Keyboard events require focus to be
        #sent to the widget, even if you
```

```
        #do not keep track of the coordinates.
    print("You clicked at", event.x, event.y)

def giveChar(event):
    #canvas.focus_set()
    print("You pressed", repr(event.char))



root = tk.Tk()
canvas = tk.Canvas(root, width = 500, height = 500)

canvas.bind("<Button-1>", giveCoords)
canvas.bind("<Key>", giveChar)

canvas.pack()


root.mainloop()

##########################################################################
##########################################################################
#Using class to click and type


from tkinter import*


class MyObj:

    def __init__(self):
        self.click_event = None

    def capture_click(self, event):
        canvas.focus_set()
        self.click_event = (event.x, event.y)

    def giveChar(self, event):
        canvas.create_text(self.click_event ,text= str(event.char)
            )
```

```
root = Tk()
canvas = Canvas(root, width = 500, height = 500)


obj = MyObj()
canvas.bind("<Button-1>", obj.capture_click)
canvas.bind("<Key>", obj.giveChar)
canvas.pack()

root.mainloop()

#########################################################################
#########################################################################

from tkinter import Tk, Canvas

def callback(event):
    draw(event.x, event.y)

def draw(x, y):
    paint.coords(circle, x-10, y-10, x+10, y+10)

def sayHi(event):
    paint.create_text(event.x, event.y, text="Hi")


root = Tk()
paint = Canvas(root)
paint.bind('<B1-Motion>', callback)
#Try this instead
#paint.bind('<Button-1>', callback)
#Or, try this instead
#paint.bind('<Motion>', callback)
#Or
#paint.bind('<Button-1>', sayHi)
#Or
#
```

```
paint.pack()

circle = paint.create_oval(0, 0, 0, 0)
root.mainloop()
```

```
#####################################################################
#####################################################################
```

```
from tkinter import *
root = Tk()
canvas = Canvas(root, width=400, height=200)
canvas.pack()
canvas.create_oval(10, 10, 110, 60, fill="grey")
canvas.create_text(60, 35, text="Oval")
canvas.create_rectangle(10, 100, 110, 150, outline="blue")
canvas.create_text(60, 125, text="Rectangle")
canvas.create_line(60, 60, 60, 100, width=3)


class MouseMover():
  def __init__(self):
    self.item = 0;
    self.previous = (0, 0)

  def select(self, event):
    widget = event.widget    # Get handle to canvas
    # Convert screen coordinates to canvas coordinates
    xc = widget.canvasx(event.x);
    yc = widget.canvasx(event.y)
    self.item = widget.find_closest(xc, yc)[0]  # ID for closest
    self.previous = (xc, yc)
    print((xc, yc, self.item))

  def drag(self, event):
    widget = event.widget
    xc = widget.canvasx(event.x);
    yc = widget.canvasx(event.y)
```

```
        canvas.move(self.item, xc-self.previous[0], yc-self.previous
            [1])
        self.previous = (xc, yc)


# Get an instance of the MouseMover object
mm = MouseMover()

# Bind mouse events to methods (could also be in the constructor)
canvas.bind("<Button-1>", mm.select)
canvas.bind("<B1-Motion>", mm.drag)

##########################################################################
##########################################################################


from tkinter import *

def giveCoords(event):
    print("You clicked at", event.x, event.y)


def draw(canvas, width, height):
    oval = canvas.create_oval(10,20,150,200, fill = 'pink', tags='
        oval')
    canvas.tag_bind('oval', "<ButtonPress-1>",giveCoords)
    #<Button-1> is a specific event.

def runDrawing(width=300, height=300):
    root = Tk()
    canvas = Canvas(root, width=width, height=height)
    canvas.pack()#Tells the window to pack
                 #the canvas within it.
    draw(canvas, width, height)
    root.mainloop()
    print("Bye!")

runDrawing(500, 500)
```

```
#####################################################################
#####################################################################

#The module ttk makes all the widgets look good.
#Try the following

from tkinter import *

window = Tk()

my_label = ttk.Label(window, text="Hello World!")
my_label.grid(row=1,column =1)

window.mainloop()

#####################################################################
#####################################################################

#These are Messages to display information to a user.
#This method returns a string which is typically ignored.
#There are three types of messages.

#Try each one, one after the other.  See what happens if you
#try them all at once.

from tkinter import messagebox

messagebox.showinfo("Information", "We are in BCC now.")
messagebox.showinfo("Error", "This is an error. Try again.")
messagebox.showwarning("Warning", "Water is hot. Don't touch.")

#####################################################################
#####################################################################

#Here are some simple yes/no type questions
#The return values are Boolean. If "cancel" is an option and the
#user choses "cancel" button, then None is returned.
```

```
from tkinter import messagebox

answer = messagebox.askokcancel("Question","Do you want to open
    this file?")
answer = messagebox.askretrycancel("Question", "Do you want to try
     that again?")
answer = messagebox.askyesno("Question","Do you like Python?")
answer = messagebox.askyesnocancel("Question", "Continue playing?"
    )

#####################################################################
#####################################################################
#This is a simple dialog object which is meant
#to ask the user for a single data value, either a
#string, integer, or floating point value.
#If the user hits "Cancel", then None is returned.

import tkinter as tk
from tkinter import simpledialog

application_window = tk.Tk()

answer = simpledialog.askstring("Input", "What is your first name?
    ",
                                parent=application_window)
if answer is not None:
    print("Your first name is ", answer)
else:
    print("You don't have a first name?")

answer = simpledialog.askinteger("Input", "What is your age?",
                                 parent=application_window,
                                 minvalue=0, maxvalue=100)
if answer is not None:
    print("Your age is ", answer)
else:
    print("You don't have an age?")

answer = simpledialog.askfloat("Input", "What is your salary?",
                               parent=application_window,
```

```
                                        minvalue=0.0, maxvalue=100000.00)
if answer is not None:
    print("Your salary is ", answer)
else:
    print("You don't have a salary?")

##########################################################################
##########################################################################
'''
Widgets and their Purposes
tk.Button, ttk.Button   Execute a specific task; a do this now
   command.
tk.Menu Implements toplevel, pulldown, and popup menus.
ttk.Menubutton  Displays popup or pulldown menu items when
   activated.
tk.OptionMenu   Creates a popup menu, and a button to display it.
tk.Entry, ttk.Entry     Enter one line of text.
tk.Text Display and edit formatted text, possibly with multiple
   lines.
tk.Checkbutton, ttk.Checkbutton Set on-off, True-False selections.
tk.Radiobutton, ttk.Radiobutton Allow one-of-many selections.
tk.Listbox      Choose one or more alternatives from a list.
ttk.Combobox    Combines a text field with a pop-down list of
   values.
tk.Scale, ttk.Scale     Select a numerical value by moving a
   slider along a scale.


There are three layout managers in the Tkinter module:

Layout Managers and their Description
place   You specify the exact size and position of each widget.
        Example:  label.place(x=20,y=30), or button.place(x=150,y
           =50)

pack    You specify the size and position of each widget relative
   to each other.
        Example: closeButton.pack(side=RIGHT, padx=5, pady=5)
```

```
grid     You place widgets in a cell of a 2-dimensional table
    defined by rows and columns.
         Example: okButton.grid(row=1, column=2).


'''


################################################
################################################
################################################


import tkinter as tk
from tkinter import ttk

def main():
    # Create the entire GUI program
    program = CounterProgram()

    # Start the GUI event loop
    program.window.mainloop()

class CounterProgram:

    def __init__(self):
        self.window = tk.Tk()
        self.my_counter = None  # All attributes should be
            initialize in init
        self.create_widgets()

    def create_widgets(self):
        self.my_counter = ttk.Label(self.window, text="0")
        self.my_counter.grid(row=0, column=0)

        increment_button = ttk.Button(self.window, text="Add 1 to
            counter")
        increment_button.grid(row=1, column=0)
        increment_button['command'] = self.increment_counter
```

```
        quit_button = ttk.Button(self.window, text="Quit")
        quit_button.grid(row=2, column=0)
        quit_button['command'] = self.window.destroy

    def increment_counter(self):
        self.my_counter['text'] = str(int(self.my_counter['text'])
            + 1)

if __name__ == "__main__":
    main()

    #You could avoid using ttk. See below.

################################################
################################################
################################################


from tkinter import *

def main():
    program = CounterProgram()

    program.window.mainloop()

class CounterProgram:

    def __init__(self):
        self.window = Tk()
        self.my_counter=None
        self.create_widgets()

    def create_widgets(self):
        self.my_counter=Label(self.window, text='0')
        self.my_counter.grid(row=0,column=0)

        increment_button = Button(self.window, text="Add 1",
            command=self.increment_counter)
        increment_button.grid(row=1,column=0)
```

```
        quit_button = Button(self.window,text="Quit", command=self
            .window.destroy)
        quit_button.grid(row=2,column=0)

    def increment_counter(self):
        self.my_counter['text'] = str(int(self.my_counter['text'])
            + 1)



main()

##################################################
##################################################
##################################################




######################################################################
######################################################################
#Here are various reliefs used for buttons.
#Note, relief does not work on certain OS.

from tkinter import *
# from tkinter.ttk import *


root = Tk()
root_width, root_height = 200, 500
root.geometry("{}x{}".format(root_width, root_height))

b1 = Button(root, text="FLAT", relief=FLAT)
b2 = Button(root, text="RAISED", relief=RAISED)
b3 = Button(root, text="SUNKEN", relief=SUNKEN)
b4 = Button(root, text="GROOVE", relief=GROOVE)
b5 = Button(root, text="RIDGE", relief=RIDGE)

b1.pack()
```

```
b2.pack()
b3.pack()
b4.pack()
b5.pack()

root.mainloop()

##########################################################################
##########################################################################

'''In what follows, you will encounter the following:

sticky
When the widget is smaller than the cell, sticky is used to
   indicate which sides and
corners of the cell the widget sticks to. The direction is defined
    by compass
directions: N, E, S, W, NE, NW, SE, and SW and zero. These could
   be a string concatenation,
for example, NESW make the widget take up the full area of the
   cell.



'''
import tkinter as tk
from tkinter import ttk
from tkinter import filedialog


class Counter_program():
    def __init__(self):
        self.window = tk.Tk()
        self.window.title("tk Examples")
        self.create_widgets()

        self.radio_variable = tk.StringVar()
        self.combobox_value = tk.StringVar()

    def create_widgets(self):
```

```
        # Create some room around all the internal frames
        self.window['padx'] = 5
        self.window['pady'] = 5

        # - - - - - - - - - - - - - - - - - - - - - - -
        # The Commands frame
        # cmd_frame = ttk.LabelFrame(self.window, text="Commands",
            padx=5, pady=5, relief=tk.RIDGE)
        cmd_frame = ttk.LabelFrame(self.window, text="Commands",
            relief=tk.RIDGE)
        cmd_frame.grid(row=1, column=1, sticky=tk.E + tk.W + tk.N
            + tk.S)

        button_label = ttk.Label(cmd_frame, text="tk.Button")
        button_label.grid(row=1, column=1, sticky=tk.W, pady=3)

        button_label = ttk.Label(cmd_frame, text="ttk.Button")
        button_label.grid(row=2, column=1, sticky=tk.W, pady=3)

        menu_label = ttk.Label(cmd_frame, text="Menu (see examples
            above)")
        menu_label.grid(row=3, column=1, columnspan=2, sticky=tk.W
            , pady=3)

        my_button = tk.Button(cmd_frame, text="do something")
        my_button.grid(row=1, column=2)

        my_button = ttk.Button(cmd_frame, text="do something")
        my_button.grid(row=2, column=2)

        # - - - - - - - - - - - - - - - - - - - - - - -
        # The Data entry frame
        entry_frame = ttk.LabelFrame(self.window, text="Data Entry
            ",
                                     relief=tk.RIDGE)
        entry_frame.grid(row=2, column=1, sticky=tk.E + tk.W + tk.
            N + tk.S)

        entry_label = ttk.Label(entry_frame, text="ttk.Entry")
        entry_label.grid(row=1, column=1, sticky=tk.W + tk.N)
```

```
        text_label = ttk.Label(entry_frame, text="tk.Text")
        text_label.grid(row=2, column=1, sticky=tk.W + tk.N)

        scale_label = ttk.Label(entry_frame, text="tk.Scale")
        scale_label.grid(row=4, column=1, sticky=tk.W)

        scale_label2 = ttk.Label(entry_frame, text="ttk.Scale")
        scale_label2.grid(row=5, column=1, sticky=tk.W)

        my_entry = ttk.Entry(entry_frame, width=40)
        my_entry.grid(row=1, column=2, sticky=tk.W, pady=3)
        my_entry.insert(tk.END, "Test")

        my_text = tk.Text(entry_frame, height=5, width=30)
        my_text.grid(row=2, column=2)
        my_text.insert(tk.END, "An example of multi-line\ninput")

        my_spinbox = tk.Spinbox(entry_frame, from_=0, to=10, width
            =5, justify=tk.RIGHT)
        my_spinbox.grid(row=3, column=2, sticky=tk.W, pady=3)

        my_scale = tk.Scale(entry_frame, from_=0, to=100, orient=
            tk.HORIZONTAL,
                                width=8, length=200)
        my_scale.grid(row=4, column=2, sticky=tk.W)

        my_scale = ttk.Scale(entry_frame, from_=0, to=100, orient=
            tk.HORIZONTAL,
                                 length=200)

        my_scale.grid(row=5, column=2, sticky=tk.W)

    #-----------------------
        # The Choices frame
        switch_frame = ttk.LabelFrame(self.window, text="Choices",
             relief=tk.RIDGE, padding=6)
        switch_frame.grid(row=2, column=2, padx=6, sticky=tk.E +
            tk.W + tk.N + tk.S)
```

```
checkbox_label = ttk.Label(switch_frame, text="ttk.
    Checkbutton")
checkbox_label.grid(row=1, rowspan=3, column=1, sticky=tk.
    W + tk.N)

entry_label = ttk.Label(switch_frame, text="ttk.
    Radiobutton")
entry_label.grid(row=4, rowspan=3, column=1, sticky=tk.W +
     tk.N)

checkbutton1 = ttk.Checkbutton(switch_frame, text="On-off
    switch 1")
checkbutton1.grid(row=1, column=2)
checkbutton2 = ttk.Checkbutton(switch_frame, text="On-off
    switch 2")
checkbutton2.grid(row=2, column=2)
checkbutton3 = ttk.Checkbutton(switch_frame, text="On-off
    switch 3")
checkbutton3.grid(row=3, column=2)

self.radio_variable = tk.StringVar()
self.radio_variable.set("0")

radiobutton1 = ttk.Radiobutton(switch_frame, text="Choice
    One of three",
                               variable=self.
                                   radio_variable, value="0
                                   ")
radiobutton2 = ttk.Radiobutton(switch_frame, text="Choice
    Two of three",
                               variable=self.
                                   radio_variable, value="1
                                   ")
radiobutton3 = ttk.Radiobutton(switch_frame, text="Choice
    Three of three",
                               variable=self.
                                   radio_variable, value="2
                                   ")
radiobutton1.grid(row=4, column=2, sticky=tk.W)
radiobutton2.grid(row=5, column=2, sticky=tk.W)
```

```
radiobutton3.grid(row=6, column=2, sticky=tk.W)

#----------------------------
# The Choosing from lists frame
fromlist_frame = ttk.LabelFrame(self.window, text="
    Choosing from a list",
                                        relief=tk.RIDGE)
fromlist_frame.grid(row=1, column=2, sticky=tk.E + tk.W +
    tk.N + tk.S, padx=6)

listbox_label = tk.Label(fromlist_frame, text="tk.Listbox"
    )
listbox_label.grid(row=1, column=1, sticky=tk.W + tk.N)

combobox_label = tk.Label(fromlist_frame, text="ttk.
    Combobox")
combobox_label.grid(row=2, column=1, sticky=tk.W + tk.N)

my_listbox = tk.Listbox(fromlist_frame, height=4)
for item in ["one", "two", "three", "four"]:
    my_listbox.insert(tk.END, "Choice " + item)
my_listbox.grid(row=1, column=2)

self.combobox_value = tk.StringVar()
my_combobox = ttk.Combobox(fromlist_frame, height=4,
    textvariable=self.combobox_value)
my_combobox.grid(row=2, column=2)
my_combobox['values'] = ("Choice one", "Choice two", "
    Choice three", "Choice four")
my_combobox.current(0)

#----------------------------
# Menus
menubar = tk.Menu(self.window)

filemenu = tk.Menu(menubar, tearoff=0)
filemenu.add_command(label="Open", command=filedialog.
    askopenfilename)
filemenu.add_command(label="Save", command=filedialog.
    asksaveasfilename)
```

```
        filemenu.add_separator()
        filemenu.add_command(label="Exit", command=self.window.
            quit)
        menubar.add_cascade(label="File", menu=filemenu)

        self.window.config(menu=menubar)

        #----------------------
        # Quit button in the lower right corner
        quit_button = ttk.Button(self.window, text="Quit", command
            =self.window.destroy)
        quit_button.grid(row=1, column=3)

# Create the entire GUI program
program = Counter_program()

# Start the GUI event loop
program.window.mainloop()

##########################################################################
##########################################################################

#Simple Interest Calculator

import tkinter as tk
from tkinter import ttk

class Counter_program():
    def __init__(self):
        self.window = tk.Tk()
        self.window.title("Simple Interest Calculator")
        self.create_widgets()



    def create_widgets(self):
        self.window['padx'] = 5
        self.window['pady'] = 5

        main_frame = ttk.LabelFrame(self.window, text='', relief =
            tk.RIDGE)
```

```
        main_frame.pack()

        principal_label = ttk.Label(main_frame, text = "Principal
            is $ ")
        principal_label.grid(row = 1, column = 1)

        self.principal_entry = ttk.Entry(main_frame,width = 40)
        self.principal_entry.grid(row = 1, column = 2)

        rate_label = ttk.Label(main_frame, text = "Annual Rate of
            Interest in % is ")
        rate_label.grid(row = 2, column = 1)

        self.rate_entry = ttk.Entry(main_frame,width = 40)
        self.rate_entry.grid(row = 2, column = 2)

        numYears_label = ttk.Label(main_frame, text = "Number of
            years is ")
        numYears_label.grid(row = 3, column = 1)

        self.numYears_entry = ttk.Entry(main_frame,width = 40)
        self.numYears_entry.grid(row = 3, column = 2)

        button_calc = ttk.Button(main_frame, text = "Simple
            Interest Calculator", command = self.interest)
        button_calc.grid(row=4,column = 1, columnspan = 2)

        inform_label = ttk.Label(main_frame, text = "Simple
            Interest earned is $")
        inform_label.grid(row=5, column =1)

        self.interest_label = ttk.Label(main_frame, text ='')
        self.interest_label.grid(row=5,column =2)

    def interest(self):
        p = float(self.principal_entry.get())
        r = float(self.rate_entry.get())/100
        t = float(self.numYears_entry.get())
        outputString = "{0:0.2f}".format(p*r*t)
        self.interest_label['text'] = outputString
```

```
# Create the entire GUI program
program = Counter_program()

# Start the GUI event loop
program.window.mainloop()
```

(1) Draw an activity diagram for the following code, and write down its output:

```
x,y=3,2
while (x+y)<18:
    print(x,y)
    x = x+3
    y =y+1
```

(2) Draw a sequence diagram for the following code:
- Chad is an instance of the class **LibraryClient**.
- Chad adds *fiction1* to his cart, by invoking the method *chad.add(fiction1)*.
- The book *fiction1*'s availability is checked, by using the method *fiction1.available()* which returns True value.
- Update *chad.bookList* by *chad.updateBookList(nonfiction2)*.
- Chad adds *nonfiction1* to his cart, by invoking the method *chad.add(nonfiction1)*.
- The book *nonfiction1*'s availability is checked, by using the method *nonfiction1.available()* which returns False value.
- Chad adds *nonfiction2* to his cart, by invoking the method *chad.add(nonfiction2)*.
- The book *nonfiction2*'s availability is checked, by using the method *nonfiction2.available()* which returns True value.
- Update *chad.bookList* by *chad.updateBookList(nonfiction2)*.

(3) Design the class, Card. Each instance of a card has the following attributes:
- a *suit* (Spade, Club, Heart, Diamond),
- a *value* (Ace, King, Queen, Jack, Two through Ten),
- accessor method *getSuit*,
- accessor method *getValue*,
- mutator method *changeSuit* which changes the suit of your card randomly, and
- mutator method *changeValue* which changes the value of your card randomly.

All the attributes are public. Draw a class diagram of your class, and write a code to implement this class.

(4) Design the class, *Cone*. Each instance of the cone has the following attributes:
- *baseRadius*;
- *height*;
- accessor method *getbaseRadius*;
- accessor method *getHeight*;
- mutator method *changeBaseRadius (r)* which changes the base radius to r;
- mutator method *changeHeight (h)* which changes the height to h;
- method *getSurfaceArea* which returns the surface area of the cone ($SA = \pi \cdot radius \cdot (radius + \sqrt{radius^2 + height^2})$);
- method *getVolume* which returns the volume of the cone ($SA = \frac{\pi \cdot radius^2 \cdot height}{3}$).

(5) Chapter 1 Review from the textbook.

(6) Write a programme that asks the user to write date in the format dd/mm/yyyy and returns Date-Month-Year. For instance, if the user enters 23/04/2019 then the programme should return 23-April-2019.

(7) Write a programme that asks the user to provide a positive integer, and returns a string where every even digit is replaced by * and every odd digit is replaced by an @. For instance, the number 325348 should return '@*@**'.

(8) Write an interactive programme which asks the user what presents they would like for their birthday. Make sure that there is a way for the user to be done listing their wishes. Then print out the user's wish-list.

(9) Chapter 2 Review from the textbook.

(10) Chapter 3 Review from the textbook (Alternatively, go through your TKinter lectures notes).

(11) Write a programme that asks the user to enter a string. Now print the string in the form of steps. For instance, if the user enters "abcd", the output should be

a
ab
abc
abcd

(12) The Euler number e is equal to the infinite series $\sum_{k=0}^{\infty} \frac{1}{k!}$. Write a function *eApproximate(n)* which for each integer $n \geq 0$ returns an approximation of e given by $\sum_{k=0}^{n} \frac{1}{k!}$.

(13) The mathematical constant π is equal to the infinite series $\pi = 4 \cdot \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right)$. Write a function *piApproximate(n)* which returns an n-th approximation $4 \cdot \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots (-1)^n \frac{1}{2n+1}\right)$.

(14) An integer in base 6 uses digits $0, 1, 2, 3, 4, 5$. Write a function *intNum(baseSix)* which takes in a string in base six, and returns its integer value. For instance, intNum(123) should return 51.

(15) An integer in base 16 uses digits $0, 1, 2, \ldots, 9, A, B, C, D, E, F$. Write a function *intNum(baseSixteen)* which takes in a string in base sixteen, and returns its integer value. For instance, intNum(F1) should return 241.

(16) Chapter 4 Review from the textbook.

(17) Write the function *kthDigit(n, k)* that takes a possibly-negative int n and a non-negative int k, and returns the kth digit of n, starting from 0, counting from the right. So *kthDigit(789, 0)* returns 9, *kthDigit(789, 2)* returns 7, *kthDigit(789, 3)* returns 0, and *kthDigit(-789, 0)* returns 9. Use *try-except* approach to avoid inappropriate inputs.

(18) Write a programme to determine whether a positive integer is prime. Draw an activity diagram to explain your programme.

(19) Write the function *mostFrequentLetter(s)* that takes a string s and returns the letter, in capital form, that occurs the most frequently in it. Your test should be case-insensitive, to "A" and "a" are the same. If there is a tie, you should return a string with all the most frequent letters in alphabetic order. For example, *mostFrequentLetter('This is a beautiful day')* must return 'AI'.

(20) Write the function *subChars(s1,s2)* which returns True if every character in string s1 is also in string s2 at least once. Note, this function should be case-sensitive; for instance, *subChars('a', 'AA')* should return False. Further, if string s1 is the empty string, then *subChars(s1,s2)* must return True. Use this function to write a function *sameChars(s1,s2)* which returns True if every character in string s1 is also in string s2 at least once, and vice versa.

(21) Chapter 5 Review from the textbook.

(22) Implement the class **Complex** numbers. Define the methods *__str__, __repr__, __add__, __sub__, __mul__, __truediv__,* and *norm*. All the instance variables are protected. Draw its class diagram.

(23) Implement the class **Modulo(n)** numbers of integers modulo n. Make certain that $n \geq 1$. Define the methods *__str__, __repr__, __add__, __sub__, __mul__* and *__truediv__*. Note that *__truediv__* could possibly return undefined results. All the instance variables are protected. Draw its class diagram.

(24) Implement the class **Polynomial(L)** of polynomial in variable x with coefficients being entries of the list L. If the list L is empty, then the polynomial is the zero polynomial. Define the methods *__str__, __repr__, __add__, __sub__, __mul__*, *derivative*, and *integrate*. All the instance variables are protected. Draw its class diagram.

(25) Chapter 6 Review from the textbook.

(26) A **pallindromic prime** is a prime number that is also a palindromic positive integer. Write a function *nthPallindromicPrime* which returns the n-th pallindromic prime for a positive intenger n. Note, *nthPallindromicPrime(1)* should return 2. The first few pallindromic primes are:

$$2, 3, 5, 7, 11, 101, 131, \ldots$$

Use try-except approach to deal with inappropriate input.

(27) Write a program which takes as input a positive integer n, and returns the n-th happy prime.

(28) Write a program which takes as input a list of positive integers, and prints out information whether the number is one, or a composite number, or a prime number, or a pallindromic integer, or a happy number, or a pallindromic prime, or a happy prime, or a happy pallindromic prime.

(29) Chapter 7 Review in teh textbook.

(30) Write a programme which accepts a text file and prints the number of characters, number of words, and number of lines in the file.

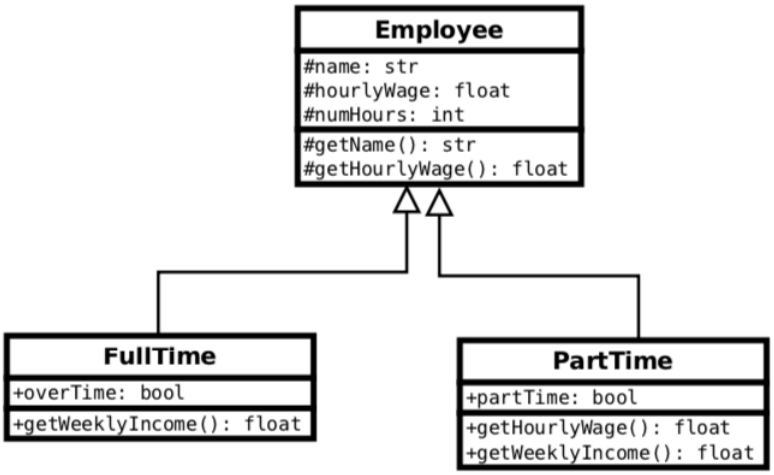
(31) Write a programme which accepts a text file with no punctuations, reads it, and writes a new file with every line reversed. For instance, if the file accepted has a line "Welcome to my home", the new file should have a line "home my to Welcome".

(32) Chapter 8 Review from the textbook.

(33) Draw a class diagram depicting inheritance in the following set-up:
   (a) All the attributes are public.
   (b) Base class is **Polygon** with attributes *numSides* (number of sides, an integer greater than 2), *isRegular* (a boolean value, stating whether the polygon is regular), and *sideLength* (a list of side-lengths). This class supports two methods, one *getNumSides()* (an accessor), and the other *perimeter()* which represents the perimeter of the polygon.
   (c) The child class is **Triangle** is obtained by setting *numSides* to 3, and this class augments the **Polygon** class by a method *getArea()* using Heron's formula for area of a triangle. The area of a triangle with side-lengths a, b, c is obtained as follows:
   Step 1: First calculate $s = (a + b + c)/2$, (half the perimeter);
   Step 2: Now, the area $A = \sqrt{s(s-a)(s-b)(s-c)}$.

   Now write a code to implement the classes.

(34) Implement the following classes using inheritance, augmentation, and overriding as follows:



   (a) Here, *name* represents the full name, *hourlyWage* represents the hourly wage, and *numHours* represents the number of hours worked by the **Employee** instance. The instance methods *getName()* and *getHourlyWage()* are accessors for the **Employee** instance.
   (b) A **FullTime** instance is expected to have *numHours* ≥ 40 per week. For every over-time hour, the employee gets paid 120% of their regular hourly wage. Define the method *getWeeklyIncome()* accordingly.
   (c) A **PartTime** instance is expected to have *numHours* < 40 per week, and will get paid only 85% of their regular hourly wage. Override the method *getHourlyWage()* accordingly for this class. Further, define *getWeeklyIncome()* for this class accordingly.

(35) Chapter 9 Review from the textbook.

(36) Rewrite the DNA to RNA transcription of section 4.2 (page 135 of the textbook) using recursion.

(37) **Draw a trace:** Problem 11.27 from page 395 of the textbook.

(38) **Draw a trace:** Problem 11.28 from page 395 of the textbook.

(39) Write a recursive function *reverseConcatenate* that takes as input a list with string entries and returns one long string which is the concatenation of all the string entries in reverse order. For instance, *reverseConcatenate(['how', 'are', 'you?'])* should return *'you?arehow'*.

(40) Solve the Towers of Hanoi problem.

(41) **TKinter :** Draw a tree using recursion.

(42) **TKinter :** Draw a pyramid (one rectangle in each level) using recursion.

(43) Chapter 11 Review from the textbook.

(44) Write an interactive function which asks the user for a name, pet's name, and the age of the pet. This function should return a list named *patients* in which each entry is a tuple (name, pet's name, pet's age).

(45) Write an interactive function which asks the user for a name, and some names of friends. This function will return a dictionary named *friends* which associates to a name-key a set of friends. For instance, here is an interaction:

```
>>> Please give me a name (<Enter> for done):  Jane
>>> Give the name of one of Jane's friend (<Enter> for done):  Holly
>>> Give the name of one of Jane's friend (<Enter> for done): Hugh
>>> Give the name of one of Jane's friend (<Enter> for done):
>>> Please give me a name (<Enter> for done): Joe
>>> Give the name of one of Joe's friend (<Enter> for done): Matt
>>> Give the name of one of Joe's friend (<Enter> for done):
>>> print(friends) #This should return a dictionary looking like
>>> {'Jane': {'Holly', 'Hugh'}, 'Joe':{'Matt'} }
```

(46) Chapter 12 Review from the textbook.

(47) **TKinter :** Draw a shrinking orange circle. That is, draw a large orange circle, which shrinks in size with a time delay of 0.25 seconds until it disappears.

(48) **TKinter :** Given an integer n greater than or equal to 3, draw a regular n-gon, coloured violet.

(49) **TKinter :** Draw a Stop Sign on a canvas. The Stop Sign is a regular octagon coloured red with a white "STOP" written in its center.

(50) **TKinter :** Write a GUI compound interest calculator. The relevant form:

$P =$ *Principal in dollars to be invested;*

$r =$ *yearly rate of interest written in decimal;*

$n =$ *number of times the interest is to be compounded each year;*

$t =$ *number of years the principal is invested.*

$I = P\left(1 + \frac{r}{n}\right)^{nt} - P$ *The final interest.*

(51) **TKinter :** Write a GUI programme which draws a tiny blue circle on a canvas wherever we click on the canvas.

(52) Chapter 15 Review from the textbook.

(53) Use *urllib.request* to write a short code which would download the webpage *http://www.bcc.cuny.edu/academics/learning-commons/* and write the contents to a file named *webpage.html*.

(54) Write a persistent echo server hosted on your localhost.

(55) Write a simple echo server which returns three copies of a message sent to it. Host this server on your localhost.

(56) Chapter 16 Review from the textbook.