Exceptional Control Flow

* A function does a job and *returns* a result – always an object of some type, unless the role is to generate some kind of side effect (**None**) rather than calculating and giving back a result
* There are two possible outcomes when a function is called:
  + The function will complete its job successfully and return an object of a type you expect
  + The function will fail to complete its job. Functions fail differently than they succeed in Python; rather than returning an object that indicates failure, instead *raise an exception*

We specify what should happen in a function when exceptions are raised by writing a **try** statement

* Legal combinations:
  + A **try** and a **finally** and nothing else
  + A **try**, at least one **except**, (optionally) an **else**, and (optionally) a **finally**

**Except** clauses can specify a type of except that they handle but will only execute when the type of exception matches the type

* **Except** clauses with no type listed can handle any kind of exception

The **finally** clause is used for *cleanup*

Recursion

Iterate over the elements of a list in Python using a **for** loop

* Assume we want to sum the numbers in a list whose elements are either integers or lists of integers; we can add the appropriate condition which requires that we know the function **type(x)** returns the type of the object **x**, and that we can compare types using == to see if they’re the same

“Recursion”, the function is a “recursive function”, which is a function that uses itself in its own solution

* Make the assumption that the recursive call will do what it’s suppose to do. If, under the assumption, the function works – if you work out the details on paper and, based on that assumption, the entire function will do what it’s suppose to do – then, as a general rule, the function works

Steps of a recursion function:

* When a function is called, the calling function pauses until the called function is complete
* When the called function is complete, the calling function picks up where it left off, using the result returned from the called function

Not all problems lead to a well-formulated solution through nesting; you’ll find that no combination of nested loops will ever solve your entire problem. To a point, there will exist an input that nests more deeply that your loops can

Sockets, Part 1

In Python, sockets are objects that encapsulate and hide many of the underlying details of how a program can connect directly to another program; you can use sockets to send data back and forth between programs running on the same machine

Sockets provide an abstraction of a connection between a program and some other program

* When two programs are connected via sockets, each program has a socket representing its end of the connection between them, with each socket having two *streams* available:
  + An *input stream*, which receives all of the data sent by the other program, in the order the other program sent it
  + An *output stream*, which takes any data written to it and sends it to the other program, in the order it was written

Sockets guarantee that if the data makes it across the network successfully it will be placed into the receiver’s input stream in the order it was placed into the sender’s output stream

* If one machine sends three messages - M1, M2, and M3 - the machine on the other side will receive the messages in that order
  + If M1 fails to make it across the network, neither M2 nor M3 will ever be seen
  + Python’s sockets does a variety of things like attempting to re-send lost information periodically and holding information received out of order until the information preceding it is received

Socket-based conversation:

* One program waits for another program to connect to it and respond to the request, then other initiates the connection
* *Client* = the program that initiated the conversation
* *Server* = the program that responded

With an IP address, the network can determine who should receive a message and how the message should get there

* “Loopback” addresses is a special range of IP addresses that can always be used to connect a computer to itself regardless of what its IP address is

The mechanism used to connect to a machine to the Internet is called a *port*

* A program acting as a server will register its interest in a particular port by *binding* to it
* In order for a client to initiate a connection to a server, the client will need to use not only the IP address of the machine where the server is running, but also the port that the server is listening on

Domain Name System(DNS) can tell you its IP address and is unlikely to affect your connections to programs on other machines significantly

*Firewalls* are software or hardware that restrict other computer’s access to computers behind them

* Some firewall software allows you to disallow certain kinds of outgoing connections, which can affect your ability to connect to programs running on other machines

*Router*: forms it’s own *local-area network*, or *LAN*

* As traffic flows into and out of the router, it performs a task called *network address translation*, or *NAT,* which means that it converts the internal IP addresses used by the computers to its own IP address for traffic going out, and converts its own IP address back to the “fake” IP addresses on the way back

**Socket** module can be used for two things: *listening* and *connecting*

* Listening means we wait for another program to contact us
* Connecting means we connect to another program

The data that is sent and received is made up of *bytes*

Sending bytes to the other program is done by calling the **send()** method on the socket, passing it the bytes we want to send

Receiving bytes from the other program is done by calling the **recv()** method on the socket, passing it the number bytes (at maximum) that we want

* An empty **bytes** object if the other program has specified that it will not send any more bytes (e.g. closed connection)
* A **bytes** object containing somewhere between one bye and the number of bytes we asked for

>>> import socket

>>> s = socket.socket()

>>> s.close()

Sockets, Part 2

*Lines* of text are terminated by a special sequence of characters called an *end-of-line sequence*

* Decoding he bytes we receive, turning them from bytes into strings
* Files contain bytes, but when we read from text files in Python, the file object converts between bytes and strings and finds where one line ends and another begins automatically;
  + Call **readline()**

Sockets in Python provide a **makefile()** method which you pass the same arguments you pass to the built-in **open()** function that opens files, and the object returned to you appears like a file object - it supports methods like **readline()** and **write()**

Protocols

When programs communicate across sockets, they have to agree on a *protocol,* which specifies what each program will send and expect to receive

* Both programs should implement the same protocol and each program should know its role in that protocol

Think of modules as collections of functions, classes, and constants

* Some are *public*: the ones that you expect other modules to need in order to solve problems
* Others are *private*: the ones that used as utilities within a module, to allow you to break up large, complex functions into smaller pieces
  + Programmers traditionally separates the “public” from the ”private” by prefixing the names with a single underscore (‘\_’) character
  + Download files into the same directory so that Python will be able to find them when one imports the other

Three things about a connection at any given time:

1. The socket across the connection is traveling
2. A pseudo-file object that lets us read input from that socket as though we were reading from a text file
3. A pseudo-file object that lets us write input to that socket as though we were writing to a text file

These three things need to be available to various functions in our module; it’s handy to create a kind of object to store all three, one thing to store, one thing to return, and one thing to pass as a parameter. (A namedtuple)

A class introduced a new type of object into your program

Classes

*Namedtuples:* one way in Python to collect heterogeneous data into a single object

* Each piece of data is called a *field* and the fields are identified by their names

The syntax **s.upper()** is called a *method call* on the object **s**

There is no **upper()** method in classes such as an integer, float, list or socket

A *class* is a blueprint for a new kind of object

We can define a new class by using the **class** construct in Python

* As in **if**statements and loops, **pass** is used to specify that a class is empty

**s.split(‘ ‘) == str.split(s, ‘ ‘)**

* Methods need what seems like an “extra” parameter, usually called **self**, which represents the object that the method is called on

*Constructing* an object by calling a class’ *constructor,*which we do by calling a function whose name is the name of the class

>>> x = Thing()

All objects in Python have a *class*, which specifies what kind of object they are

* All objects in Python have a collection of *attributes*
* We assign values into attributes within the class; the syntax is important
  + **Object\_name.attribute\_name = value**
* Writing a method in a class is similar to writing a function; the **def** construct is used
  + When you make calls to methods, an extra parameter is added before all the others; that parameter is called **self**
  + When we want to assign a value to an attribute: specify the object whose attribute we want to assign, followed by a dot, followed by the name of the attribute
  + A single underscore in front of the name can be used in the same way with attributes and methods in classes

Downloading data from the web

In a web page, the notion of “where you want to go” is encapsulated by a URL (*Uniform Resource Locator)*, which specifies a few things:

* What protocol should be used to download the web page?
* From what host should the web page be downloaded? Occasionally, we also specify the port, when we want it to be something other than the default
* What page on that machine should be downloaded?

http:// www.ics.uci.edu/~thornton/ics32/CodeExamples/ExceptionalControlFlow/oops.py

* The characters (preceding the colon) indicate what protocol should be used for the network conversation
* Two slashes is the host
* The rest of the URL specifies what web page to download from the given host using the given protocol

“/~thornton/ics32/CodeExamples/ExceptionalControlFlow/oops.py”

HTTP (*HyperText Transfer Protocol*) is a *request-response protocol*, which means that its conversations go like this:

* Client initiates connection to server
* Server accepts connection
* Client makes a request
* Server sends a response

GET /~thornton/ics32/CodeExamples/ExceptionalControlFlow/oops.py HTTP/1.1

Host: www. ics.uci.edu

The first line of a GET request begins with the word **GET**, followed by the web resource you want to download

The second line (and subsequent lines) are called *headers*, which allow us to specify a variety of supplementary information

The first line of response indicates that the server agrees to have a conversation, followed by a *status code*

* 200 (OK) everything went as planned
* 404 (Not Found) the server doesn’t have the page you asked to download

**Date**: the date/time at which the response was generated

**Server**: specifies what type of server is being run and what version

**Content-Length**: specifies the length, in bytes, of the content that will be sent back. This allows the client to know when the content has ended

**Content-Type:** specifies what kind of content is being sent back

The **urllib.request** module has one function that we’re interested in: **urllib.request.urlopen()**

* the **urlopen()** function returns an object called an **HTTPResponse**, which provides a few useful attributes and methods, the most important of which is the **read()** method, which retrieves all of the content from the response and returns a **bytes** object containing those contents

Chapter 7.1-7.4, Chapter 10.1-10.2

Variable names defined inside a function should be “invisible” to the calling program: They should be variables that exist only locally, in the context of the execution of the function and should not affect variables of the same name in the calling function

* Names that are assigned during the execution of a function call are said to be *local names* and they are local with respect to a function call
  + Are only visible to the code inside the function
  + Do not interfere with named defined outside of the function, even if they are the same
  + Exist only during the execution of the function; they do not exist before the function starts execution and they no longer exist after the function completes execution

Names assigned in the interpreter shell or in a module outside of any function are said to have a *global scope*

* Variables with global scope are referred to as *global variables*

Python interpreter searches for the name definition in this order:

1. First the enclosing function call namespace
2. Then the global (module) namespace
3. Finally the namespace of module *builtins*

Error objects are called *exceptions* because when they get created, the normal execution flow of the program is interrupted, and the execution switches to the so called *exceptional control flow*

* The default exceptional control flow is to stop the program and print the error message contained in the exception object

The *try* and *except* statements work in tandem

* If no errors occur, then the code block below *except* is ignored
* However, if an exception is raised during the execution of a *try* code block, the python interpreter will skip the execution of the remaining statements in the *try* code block and execute the code block of the *except* statement instead

Format of try/except pair of statements is:

*try:*

*<indented code block 1>*

*except:*

*<indented code block 2>*

*<non-indented statement>*

<*indented code block 1>* is attempted first and if it passes, then <*indented code block 2>* is ignored and continues with <*non-indented statement>*

If an exception is raised during <*indented code block 1>*, then the remaining statements in *<indented code block 1>* are not executed and <*indented code block 2>* is executed, and if no new exception is raised, then the execution continues with the <*non-indented statement*>

When the module is executed (imported), then the module is (also) a namespace

* This namespace has a name, which is the name of the module
* In this namespace lives the names that are defined in the global scope of the module: the names of the functions, values, and classes defined in the module (the module’s *attributes*)

When the Python interpreter executes an *import* statement, it:

1. Looks for the file corresponding to the module
2. Runs the module’s code to create the objects defined in the module
3. Creates a namespace where the names of these objects will live

*Base case* of the recursion: the condition that will ensure that the recursive function is not going to call itself forever

FINAL REVIEW

Classes, Duck Typing, and Interfaces

Python uses a technique called *duck typing* when deciding what we can and can’t do with the values stored in variables

* If you try to call a method on an object it’s legal as long as that object’s class has a method; otherwise it is illegal
* Sometimes, the same method or same operator will behave wildly differently depending on the type of objects its called on, but the behavior will always be the “right” behavior for that type, without having to do anything special
  + Length of a string = number of characters
  + Length of list = number of elements
  + Adding two integers = sum
  + Adding two lists = gives their concatenation

def foo(x, y):

return x.bar(y) \* 2

* **x** must be an object of some class that has a method called **bar** that takes one para
* meter (**self**)
* **y** must have type that is compatible as an argument to **bar**. Depending on **x**’s type and depending on what its **bar** method does, the constraint will be different
* The type of value return from **bar** must be something that can be multiplied by 2
* The type of value retuned from **foo** is whatever type of value you get when you multiply **bar**’s result by 2

Duck typing is advantageous because we can write multiple classes and intentionally give them the same *interface* (ie., they each have one or more methods in common, whose signatures and meanings are the same), then use objects of the classes interchangeable

Two-Dimensional List Algorithms

When assigning an object into a variable in Python, the variable doesn’t actually store the object; instead the variable stores a *reference* to the object (similar to the location in memory where the object resides)

* You can get the value of the reference by calling the function **id**

Mutable types are ones whose objects can be changed; immutable types are the ones whose objects cannot

* Integer values are immutable in Python; nothing you can do to change an integer object once it’s created
* Lists are mutable, which means that alterations to an existing list *change that list* as opposed to building a new one
* The **id** of a list does not change when you append something to it (+= does too)
  + The + operator does not; it builds a new list that is the concatenation of the two lists being added together
* Mutating one list also mutates another

When iterating over a list **a**

* Use a **for** loop that iterates over a list’s elements **(for x in a):**
  + Variable x refers to each object. X refers directly to the objects in the list, those objects are immutable therefore modifying **x** will have no effect on the list
* A for loop that iterates over the *indices* of the list’s elements (**for x in range(len(a))**):\_
  + Variable **x** does not refer directly to each object in the list; it stores the *index* of each element
* A **while** loop: manages the values of variables that access indices manually, ensuring that they remain valid

Basics of the “tkinter” Library

Building a GUI: create a completely empty GUI window using tkinter

>>> import tkinter

>>> window = tkinter.Tk()

>>> window.mainloop()

* Creating a **tkinter.Tk** object creates a **tkinter-**based GUI, which causes a window to pop up on your screen; however, that window will be inactive
* **Tkinter**-based GUIs are built using event-based programing

We make a **tkinter** GUI active by calling the **mainloop()** method on its main window

A **tkinter** GUI has a *main loop* (sometimes called *event loop*) that watches for a variety of inputs

The **mainloop()** method causes your own code to stop making progress; the loop doesn’t end until the window is dismissed

* *Event-based programming* : When an event happens, a function or method in the code will be called to process that event; what that function or method returns, **tkinter’s** main loop goes back to watching for inputs
  + In a program with a **tkinter**-based GUI, we don’t handle all control flow in the program; instead we create the GUI, “wire up” a set of event handlers, and the code waits for the event handles to be called

A window in a **tkinter** GUI consists of building blocks called *widgets*

Each widget has a set of *options* that can be configure when the widget is first created, by passing keyword arguments to its constructor

* *Master*: the widget in which a widget is placed
* *Text*: the text that is displayed inside the widget
* *Font*: specifies the font which the widget’s text should be displayed

**Tkinter**’s main loop watches for various inputs and converts them to *events*

* Some widgets have *behaviors* built into them
* Example: the **button** widget has an option called **command**; if you set this option to a function that takes no paramenters, the function will be called

*Event binding*: binding to an event on some widget is done by calling the **bind()** method on that widget

* The first parameter to **bind()**is the description of the event (or sequence of events) you’re interested in
  + **‘<Enter>’** is an event that’s generated when the mouse cursor moves into a widget
  + second parameter is an *event handler function*: event handler functions are required to take a single parameter, which will be passed an *event object* that describes the event

The Canvas Widget and Object-Oriented Programming Using “tkinter”

**Canvas** provides the ability to do programmatic drawing of shapes like lines, rectangles, ovals and arcs

The **canvas** widget uses the same coordinate system that is in two-dimensional computer graphics

* It’s customary for a point to be described by an (*x, y*) coordinate
* The *origin point* (0,0) is at the top-left, and x-coordinates increase as you move to the right and y-coordinates increase as you move down

Functions in python are objects; this allows us to store them in variables, pass them as parameters and return them from other functions

Classes contain methods called objects of that class

* A *bound method* is one in which the **self** parameter has already been bound to the object preceding the dot
* If we want to call a bound method, we pass it the *missing* arguments

A **tkinter**-based GUI is written using event-based programming techniques; we ask **tkinter** to handle all control flow and watch for inputs, then notify us only when events occur in which we’ve registered an interest

* Classes are a way to bring together data and the operations that know how to manipulate the data
* The objects store things and know how to do things with the data stored

If an event handler is a bound method, the **self** argument is already bound; tkinter won’t need to pass it. When tkinter calls the event handler and passes it an event object, it is calling a method that takes a self and an event object; it has access to two things:

* Information about the event that’s happening now, in form of event object
* Any information about things that have happened before, stored in attributes of self

COLORS:

* Specified as a ‘#’ character followed by three hexadecimal digit amounts
* Specifies an amount of red, green and blue, respectively

Resizing a “tkinter” Canvas

We want shapes on the canvas to change correspondingly with the size of the canvas window (we’ll need to redraw each point with new coordinates)

*Fractional coordinates*: uses floating-point numbers to specify coordinates as a fraction of the distance from the left to the right (horizontally, in x direction) and from top to bottom (vertically in y direction)

* The coordinates were independent of the size of the Canvas

Applying an alternative coordinate system

1. Know the fractional coordinates of the shapes you want to draw
2. Bind the even that occurs whenever the size of the Canvas changes
3. Convert the fractional coordinates to pixel coordinates (multiply the fractional x-coordinate by the width of the Canvas-in pixels-and multiplying fraction y-coordinate by height)

REDRAWING

Self.\_canvas.delete(tkinter.ALL)

PIXELS

Canvas\_width = self.\_canvas.winfo\_width()

Canvas.height = self.\_canvas.winfo\_height()

Building the Spots Application Using “tkinter”

*Alternative coordinate systems*: where we represent the same coordinate in different ways in different parts of the program

* Given a Canvas size, there is a one-to-one mapping between a fractional coordinate and its corresponding pixel coordinate; fractional coordinates and pixel coordinates gives two different ways of representing the same point

The Grid Layout Manager

The **grid** layout manager arranges widgets by placing each one into one or more cells on an invisible grid

* They arrange themselves into rows and columns with the widgets arranged within the cell
* When you create a GUI using the **grid** layout manager, you specify which widgets will appear in which cells, as well as how the cells’ sizes change as the window size changes

All about **grid**-based layout

* All cells in a row have same height and all cells in a column have same width
* Heights of different rows and widths of different columns can be different
* Heights of the rows and widths of columns are determined by initial size of widgets they contain
* As window’s size grows, additional space is allocated according to the *weights* specified on each
  + Weights are a relative measure
* Widgets are arranged within grid cells based on an option **sticky**
  + The **sticky** value is specified by adding directions “north”, “south”, “east”, “west”
  + By default: widgets float in center
* *Padding* is border around outer edge of grid cell
  + ***Padx*** *:* horizontal padding
  + ***Pady***: vertical padding
* A widget can span more than one row/one column by setting the **rowspan** and/or **columnspan**
  + Example: if a widget is specified to be in **row** 5 and **column** 4 with **rowspan** 3 and **columnspan** 2, it will span three rows starting from row 5 and two columns starting from column 4
* A **grid** layout can be applied to the entire contents of a window; it can also be applied to contents of a Frame widget, a white whose job is to contain other widgets

Modal Dialog Boxes in “tkinter”

Pop-up windows are often called *dialog boxes*

* *Modal dialog boxes*: requires the user to complete the task given by the dialog box before proceeding to use the application
* *Modeless dialog boxes*: pops up and allows the user to complete some task, while allowing the rest of the application to be used normally

**Mainloop** is a loop that dispatches events to any code that’s interested in them (*event loop*)

* Example : **<Motion>** (everytime the mouse is moved) this event is generated and the main event loop processes it by checking to see if the code as bound to that event
* When the window is dismissed, the main event loop ends and the call on **mainloop()** method returns

Creating a modal dialog boxes using Tkinter

* Create a **tkinter.Toplevel** object
  + A **Toplevel** object represents a *top-level window*, which is to say a window that has all the normal decorations a window has (i.e. title bar, minimize, maximize, etc)
* We add widgets to the **Toplevel** object so they appear in the new window
* We follow two-step process to give control over to new window
  + First, tell the window we want to *grab* all the events in the application; disables the normal event handling done in other parts of the application (**grab\_set**)
  + Second, start a new event loop specific to the new window; causes main event loop to pause until new event loop ends
    - This is done by using the method **wait\_window** which will effectively cause the rest of the application to *wait* until the window has been dismissed
* Usually everything is in response to an event in the root window

We’ve adopted the convention of representing the Tkinter applications as classes so the application can start by calling a **start()** method on it

The modal dialog boxes can have the same convention

* Write class that represents the modal dialog box
* If the modal dialog box needs parameters to configure it, pass them as parameters to the class’ \_\_init\_\_ method
* The \_\_**init**\_\_ method creates all the widgets, set up layouts, wire up event handlers
* The modal dialog box should be displayed from a **show()** method (very similar to a **start()**); the **show()** method won’t reutn until the modal dialog box is dismissed
* EXAMPLES FROM GREETINGS.PY
* def show(self) -> None:

# This is how we turn control over to our dialog box and make that dialog box modal

self.\_dialog\_window.grab\_set()

self.\_dialog\_window.wait\_window()

* self.\_first\_name\_entry = tkinter.Entry(master = self.\_dialog\_window, width = 20, font = DEFAULT\_FONT)
* self.\_first\_name\_entry.grid(row = 1, column = 1, padx = 10, pady = 1, sticky = tkinter.W + tkinter.E)
* self.\_last\_name\_entry = tkinter.Entry(master = self.\_dialog\_window, width = 20, font = DEFAULT\_FONT)
* self.\_last\_name\_entry.grid(row = 2, column = 1, padx = 10, pady = 1, sticky = tkinter.W + tkinter.E)
* def \_\_on\_\_ok\_\_button(self) -> None:

self.\_ok\_clicked = True

self.\_first\_name = self.\_first\_name\_entry.**get()**

self.\_last\_name = self.\_last\_name\_entry.**get()**

self.\_dialog\_window.detroy()