Python Programming

INFO.2820.061

Instructor: Andy Garside

Namespaces, Optional Function Arguments, High-Order Functions, Exceptions, and Simple Graphics using Object-based Programming

Introduction

In this lecture we will expand our understanding of Python Variables (module variables, local/temporary variables and function parameters) along with their scopes and lifetimes. We will also extend our understanding of Python functions, including optional arguments, anonymous (lambda) functions, and passing functions as arguments to functions using higher-order functions like **map**(), **filter**() and **reduce**(). We will look at how to handle exceptions in Python, and look closer at object-based programming concepts, including classes, objects and methods while investigating simple graphics.

Understanding a Program's Namespace

In any structured programming language there are a set of items whose names exist at varying points during the time a program is running. Sometimes those names exist at the beginning of a program and last for the life of the program, and sometimes those names come into existence at some point in the program and cease to exist before program termination (For example, a parameter to a function exists only when the function is called and it ceases to exist when the function ends.) The time between when a name comes into existence and when it ceases to exist is known as its **lifetime**, and the set of these names and their values are known as the **namespace**.

To understand a language, it is essential to understand the lifetimes of variables. It is also essential to understand what parts of a program can access the variables during their lifetimes (called the **scope** of a variable). In the next few sections we will look at different types of Python variables, their lifetimes and their scopes.

Module Variables, Parameters, and Temporary Variables.

Consider a Python module called fibonacci.py containing the following code:

```
fibonacci = (1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,
           610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657)

def gcd(smaller, larger):
    while smaller > 0:
        remainder = larger % smaller
        larger = smaller
        smaller = remainder
    return larger


r = []
for i in range(len(fibonacci)):
    r.append(gcd(fibonacci[i],9000000))
```

This program has a number of different categories of variables.

Module variables are variables that are defined at the outermost level of a module, and they are assigned values at the point where they first appear. You can view any module's module-level variables by importing the module and using the **dir**() function as shown below.

```
>>> import fibonacci
>>> dir(fibonacci)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'fibonacci', 'gcd', 'i', 'r']
```

The output includes variables like __file__, __package__, __name__, etcetera which are define for every module. It also includes variables from *fibonacci.py*, including **fibonacci**, **gcd**, **i** and **r**. Note that **i** is a module-level variable because it first appears at the outermost level, even though it is only used within the **for** loop. Notice too that the function name, **gcd**, is included in the output. In Python, function names are consider variables.

Parameters appear in function and method headers and behave like variables. They receive their values when the function or method is called. In *fibonaci.py*, **smaller** and **larger** are parameters to the **gcd** function.

Temporary variables typically appear in the body of a function or method, or within a loop or section statement. Like module variables, they are assigned a value at the point in the body where they first appear. The variable **remainder** in the **gcd** body is an example of a temporary variable.

Method names are names associated with object methods. For example, in fibonacci.py, the method **append** is a method of the **list** object.

Variable Scope & Lifetimes

The **scope** of a variable is the area of a program over which it can be seen. The **lifetime** of a variable is the period of time that the variable exists within that **scope**. When the variable comes into existence, storage is allocated for it; when the variable goes out of existence, storage is reclaimed by the Python Virtual Machine (PVM). Each of the variable types above have different scopes and may have different lifetimes.

Temporary variables (also called **local** variables) have a **scope** consisting only of the body in which they are defined. They cannot be seen from anywhere else within the module. For example, the **scope** of the temporary variable **remainder** is the body of function **gcd**. If the program tried to, say, print the value of **remainder** outside the function **gcd**, a Traceback would occur indicating the name **remainder** is not defined. The **lifetime** of the variable **remainder** begins when it is first assigned a value as the first line within the **while** loop. The variable continues to exist until the call to the **gcd** function returns. This means, for example, you could put the statement **print(remainder)** right before the **return** statement, outside of the while loop, and it would successfully print the last value assigned to **remainder** within the while loop. However, if you put that same statement above the while loop, you would get a Traceback stating “UnboundLocalError: local variable 'remainder' referenced before assignment”

Parameters have a **scope** of the entire function. Like the temporary/local variables, they cannot be seen from anywhere else within the module. They, effectively, behave like temporary variables whose **lifetime** begins when the function is called (at which point they are assigned the value of their corresponding arguments) and ends when the function call returns.

Module variables have a **scope** that includes the entire module below the point where the variable is first defined. However, the **scope** also includes the bodies of **functions** that occur **earlier** in the file, provided the module variable has been defined before the function is called. For example, in *fibonacci.py*, you could put the statement **print(r)** immediately before the **return** statement in the **gcd** function and it would print the value of the module variable, **r**, each time gcd is called because the statement **r = []** defines **r** before the first call to **gcd** occurs. The **lifetime** of a module variable begins when it is first defined and generally exists for the lifetime of the program that introduces or imports those module variables.

The `global` declaration

It is not always apparent to new Python programmers when Python will create a new local/temporary variable verses accessing an existing module-level variable with the same name. Some of the confusion comes from the fact that a Python module-level variable is not identical to a "global" variable in a language like C.

For example, in Python, module-level variables can be read within a function but, by default, the function cannot assign them new values. Consider the following program.

```
x = 5
y = 7

def f():
    print(y)
    x = 10
f()
print(x)
```

```
7
5
```

In the program above, the **module-level** variable, **y**, is initially assigned the value **7.** The value of **y** is then printed by the **print(y)** statement inside function **f**, resulting in the first output value, **7**.

The **module-level** variable **x** is initially assigned the value **5**. However, within the function, **f**, an attempt is made to assign **x = 10**. The issue is that Python does not see the **x** within function **f** as the module-level variable **x**, but, rather, as a temporary/local variable **x** that is automatically created by Python when the assignment **x = 10** occurs within the function. This behavior is clearly demonstrated when the **print(x)** statement is executed outside the function and the original value of **x** is displayed (which is **5)**. To avoid this issue, Python provides the **global** keyword.

Within the body of a function, a variable can be declared to reference a module-level variable, rather than a local/temporary variable, by using the keyword **global**. The program below is identical to the one above only the statement **global x** has been added directly after the definition of the function **f**.

```
x = 5
y = 7

def f():
    global x
    print(y)
    x = 10
f()
print(x)
```

```
7
10
```

Notice when this modified program is run that the result of the **print(x)** statement is now 10. This is because the **global x** statement directed the Python Virtual Machine to treat the variable **x** within the function **f** as a reference to the module-level variable **x**, rather than a local/temporary variable **x**.

Remember, the keyword **global** makes a variable's reference global only within the function in which the global statement appears. Any other function that refers to the variable will treat it as a local/temporary variable unless it, too, contains a global declaration.

Default and Optional Arguments

One of the features I personally like and frequently use in Python are optional arguments.

There are times when a function requires a number of options in order to offer full functionality. However, some of the arguments to the function may infrequently change. In this case, it is convenient if those arguments default to a value so the programmer does not have to specify those arguments every time the function is called.

Consider the following simple function named **sequence** which generates an integer sequence given a **stop** value, a **start** value, and a **step** (note: this simple function does not do any error checking).

```
def sequence(stop, start, step):
    """ Simple integer sequence generator """”
    s = []
    while (stop >= start):
        s.append(str(start))
        start += step
    print(','.join(s))
```

A typical call to this function requires three arguments, as seen below.

```
sequence(10,1,1)
1,2,3,4,5,6,7,8,9,10
```

If we assume, in most cases, that the programmer will want a **step** of 1, it would be convenient not to have to specify the step argument (that is, make specifying it's value **optional**), and for the function, **sequence**, to default to a **step** of 1. Python supports this concept using a concept called **default arguments** (also known as **keyword arguments**)

To make an argument optional in Python, you replace the parameter name in the function definition with the following syntax

<parameter-name> = <value>

For example, to make the argument **step** optional, you would rewrite the function header for **sequence** as

```
def sequence(stop, start, step=1):
```

After making this simple change, we can generate the same sequence we generated earlier with the following call

```
sequence(10,1)
1,2,3,4,5,6,7,8,9,10
```

Note that calls to **sequence** can still be made as they were before, even though the **step** argument has a default value. For example, we can specify the value 3 as the third argument to **sequence** and generate a sequence of integers with a step of 3.

```
sequence(10,1,3)
1,4,7,10
```

If we also believe that most applications will want a sequence of numbers beginning with 1, we can give the **start** parameter a **default** value as well. To do this, we change **start** to **start=1** in the function header, as follows.

```
def sequence(stop, start=1, step=1):
```

The following call will now generate the same integer sequence we generated earlier.

```
sequence(10)
1,2,3,4,5,6,7,8,9,10
```

Of course, we can override the defaults for **start** and **step** by explicitly specifying them in the argument list to our **sequence** call, as follows

```
sequence(10,0)
0,1,2,3,4,5,6,7,8,9,10
```

```
sequence(10,0,3)
0,3,6,9
```

Rules for default and optional arguments.

When working with optional arguments and default values, there are a few things to keep in mind.

1. Required arguments must be specified in a function definition **before** arguments with defaults. For example, the following is **not** a legal syntax. The programmer clearly wants the **start** value of the **sequence** function to default to 1, but doing the following results in the Python Virtual Machine returning the error “non-default argument follows default argument”

   ```
   def sequence(stop, start=1, step):
   ```

   To define **start** with a **default** of **1** the programmer must either also give **step** a default value, or reorder the parameters to the function so that **start=1** follows the **required stop** and **step** parameters.

2. When calling functions that have default arguments, the **required arguments** must **all** be specified and appear in the order they appear in the function definition. The **default arguments** may, however, be specified in two ways:

   1. **By position**. This is how we’ve specified the arguments in our examples so far. In this case, arguments that appear after the required arguments are assigned to the optional/default parameters of the function in the order they appear in the function header.

   2. **By keyword.** In this case, the programmer can selectively set individual optional parameters using the following syntax in the function call.

      <parameter-name> = <value>,…

      For example, if we want to call **sequence** and set the optional argument **step** to **3** without specifying the **start** argument, we can do the following. (The **start** argument will be set to its **default** value of 1.)

      ```
      sequence(10, step=3)
      1,4,7,10
      ```

Extending function capability with optional arguments.

Let's say a user of the **sequence** function desires to generate a sequence of integers but wants the character separating the integers to be a space rather than a comma. If we were to add a required parameter to the function definition for specifying the integer separator, any existing code that calls **sequence** would break because the existing calls would be missing the newly-added required argument. However, using a default parameter, we can extend the function definition without breaking existing programs.

Consider the following function header. It adds a parameter called **separator** to the **sequence** function definition and gives it the **default** value of a comma.

```
def sequence(stop, start=1, step=1, separator=','):
```

Within the body of the **sequence** function we can then change the print statement from this

```
print(','.join(s))
```

to this

```
print(separator.join(s))
```

and create an updated **sequence** function that looks as follows

```
def sequence(stop, start, step, separator=','):
    """ Simple integer sequence generator """”
    s = []
    while (stop >= start):
        s.append(str(start))
        start += step
    print(separator.join(s))
```

With this change, the original calls to **sequence** still work, unmodified.

```
sequence(10,1,1)
1,2,3,4,5,6,7,8,9,10
```

However, users who want to change the **separator** of the sequence can now do so by explicitly specifying the value of the optional **separator** argument. For example,

```
sequence(10,1,1,' ')
1 2 3 4 5 6 7 8 9 10

sequence(10, separator=' ')
1 2 3 4 5 6 7 8 9 10
```

Higher-Order functions: map, filter, reduce

There are many times in programming when the job of a piece of code is to call a function a number of times, each time giving it a different parameter. For example, consider the following code that converts a list of strings representing numbers to a list of equivalent integers.

```
s = ["301", "22", "13", "647"]
nums = []
for i in range(len(s)):
    nums.append(int(s[i]))

print(nums)
[301, 22, 13, 647]
```

In Python, it's possible to simplify this code using a higher-order function. A higher-order function takes a function name and a set of data values as arguments. The named function is then applied to each data value and a set of results, or a single data value is returned.

Functions as First-Class Data Objects

In order to understand higher-order functions we must first understand a bit more about how Python treats functions. In Python, functions can be **first-class data objects**. This means a function can be assigned to a variable, passed as arguments to other function calls, returned as values from other functions, and stored in data structures like lists and dictionaries. For example,

```
print(str(4))
convert_to_str = str        # convert_to_str is an alias for str
print(convert_to_str(4))    # apply convert_to_str to an argument
```

```
4
4
```

Passing a function as an argument to another function is no different than passing any other argument value. For example, below, the **round** function is passed as an argument to another function named **runit**. When **runit** is called with **round** as the first argument, the argument, **round**, is evaluated, producing the **round** function object itself, and this is bound to the associated named parameter, **f**, in the **runit** function definition. The parameter **f**, which behaves like a local/temporary variable, can then be used to invoke the **round** function along with any required arguments.

```
def runit(f,value):
    return f(value)

print(runit(round, 10.6))
print(runit(abs, -13))
```

```
11
13
```

The `map` function

The Python **map** function is an example of a higher-order function. The format of a call to the **map** function looks as follows

map(<function>, <sequence>)

The **map** function applies (maps) the <function> to each value in the <sequence> and returns a new sequence of the results. The <sequence> can be any valid Python sequence, for example, a list, tuple or string.

Below is the program we saw earlier that converts a list of strings representing numbers to a list of equivalent integers.

```
s = ["301", "22", "13", "647"]
nums = []
for i in range(len(s)):
    nums.append(int(s[i]))

print(nums)

print(nums)

[301, 22, 13, 647]
```

This program can be rewritten more simply using the **map** function as follows

```
s = ["301", "22", "13", "647"]
nums = list(map(int, s))

print(nums)

[301, 22, 13, 647]
```

Above, the **map** call passes each of the numeric strings in list **s** to the function **int**, which converts each strings to an integer and appends it to a new sequence. The **list** function then converts that sequence to a **list** and assigns the variable **nums** as the name (alias) for the list.

The `filter` function

The Python **filter** function is another example of a higher-order function. The format of a call to the **filter** function looks as follows

filter(<function>, <sequence>)

Like the **map** function, the **filter** function applies (maps) a <function> to each value in a <sequence> and returns a new sequence of the results. The difference is that with **filter**, the <function> provided to **filter** must return a Boolean value (functions of this type are called **predicate** functions, or simply, **predicates**). When the return value of <function> is **True**, the sequence value is added to the new sequence of results. Otherwise, it is not.

As with the **map** function, the <sequence> can be any valid Python sequence, for example, a list, tuple or string.

Consider the following program. It defines a simple function named **odd** that takes an integer and returns **True** if the integer is odd. Otherwise, it returns **False**.

The **for** loop generates a sequence of integers from 0 to 9 and calls the **odd** function to determine if the integer is odd. If so, it appends the integer to the list called **nums**.

```
def odd(n):
    return n % 2 == 1

nums = []
for i in range(10):
    if odd(i):
        nums.append(i)

print(nums)
```

```
[1, 3, 5, 7, 9]
```

We can simplify the **for** loop above and rewrite it in a less error-prone format using the **filter** function, as shown below.

```
def odd(n):
    return n % 2 == 1

nums = []
nums = list(filter(odd, range(10)))

print(nums)
```

```
[1, 3, 5, 7, 9]
```

In the code, the **filter** function passes each of the generated sequence values 0 through 9 to the **odd** function and, whenever the odd function returns **True**, the

value is added to the returning sequence. The **list** function then converts the sequence into a list, which is then assigned the name (alias) **nums**.

The reduce function

The **reduce** function in Python's **functools** module is yet another example of a higher-order function. The format of a call to the **reduce** function looks as follows

reduce(<function>, <sequence>)

Like the **map** and **filter** functions, the **reduce** function applies (maps) the <function> to each value in the <sequence>. The difference is that the **reduce** function returns the **accumulated values** of the function calls, rather a sequence of the results. Like the **map** and **filter** functions, the <sequence> can be any valid Python sequence, for example, a list, tuple or string.

Consider the following program. It contains a simple function, **multiply**, that takes two arguments, **x** and **y**, and multiplies them together, returning the result. The **for** loop loops through the list of integers in **nums** and multiplies each by the value in **accumulator**. When the loop is done, **accumulator** then holds the product of all the integers in **nums**.

```
def multiply(x, y):
    return x * y

nums = [1, 2, 3, 4]
accumulator = 1
for x in nums:
    accumulator = multiply(x, accumulator)

print(accumulator)
```

```
24
```

reduce expects two arguments: a function of two arguments and a sequence of values. When called, **reduce** applies the function to each value in the sequence and returns the accumulated value as a result. Using **reduce**, we can rewrite the program above as follows.

```
from functools import reduce

def multiply(x, y):
    return x * y

nums = [1, 2, 3, 4]
print(reduce(multiply, nums))
```

Creating Anonymous functions with `lambda`

In programming, it is not uncommon to see simple functions like the function **odd** below. Though these functions may be simple, they are often a useful way to abstract away an idea and isolate functionality to a single place where it can be called from any number of places and any number of times. However, there are times when such a function is only intended to be called a single time, making it undesirable to actually create a named function to perform the calculation. When this is the case, Python offers the ability to create an in-line, **anonymous** function, also call a **lambda** function.

The syntax of a **lambda** function is very specific and limited.

lambda <argument1, ..., argumentN> : <expression>

All of the code must appear on one line and the <expression> cannot contain a selection statement (because selection statements are not expressions.)

Consider the program below. It contains the same **odd** function we defined earlier. In this case, the **for** loop iterates through all the integers in **nums**, calling the **odd** function with each integer. If the integer is odd, the Boolean value **True** is added to the list **is_odd**. Otherwise, the Boolean value **False** is added. When the loop completes, the contents of the **is_odd** list are displayed.

```
def odd(n):
    return n % 2 == 1

nums = [1, 2, 3, 4]
is_odd = []
for i in nums:
    if odd(i):
```

```
        is_odd.append(True)
    else:
        is_odd.append(False)

print(is_odd)
```

```
[True, False, True, False]
```

Using an anonymous lambda function, we can replace all of the code above with the two lines below. As defined, the lambda function takes a single argument, **n**, and evaluates the expression **n % 2 == 1**, returning **True** if **n** is odd and **False** otherwise. The map function then applies the lambda function to each element in **nums**, returning a new sequence containing the results of the calls to the lambda function. The sequence is then converted to a list using the **list** function and the results are displayed with the **print** function.

```
nums = [1, 2, 3, 4]
print(list(map(lambda n : n % 2 == 1, nums)))
```

```
[True, False, True, False]
```

Though lambda function are limited they do have their uses, and you will certainly find them in existing Python modules.

Exceptions

There are different kinds of errors in programming.

1. There are **logic errors**, which essentially equate to writing a program that compiles and runs to completion, but it doesn't contain the correct sequence of steps to solve the problem.
2. There are **syntax errors**, which are, effectively, malformed language statements. In Python, syntax errors are reported by the PVM when a program is compiled into byte-code, resulting in a printed error message and a halt to program execution. For example,

   ```
   >>> x = 45 + * 3
   SyntaxError: invalid syntax
   ```

3. There are **semantic (runtime) errors**, which occur when a **syntactically** correct program statement is executed and the result is **invalid**. Division by 0 is a good example,

   ```
   >>> x = 2 - 2
   >>> 5 / x
   Traceback (most recent call last):
     File "<pyshell#70>", line 1, in <module>
       5 / x
   ZeroDivisionError: division by zero
   ```

In Python, when a **semantic** error occurs, execution is halted and a **Traceback** is displayed indicating the cause of the error. This process is known as raising an **exception**.

While some **exception**s can be prevented (for example, in the above code, we could check that x is not zero before performing the division), other's cannot be so easily avoided.

Consider the following simple program that calculates an employee's pay. It prompts the user for the number of hours the employee worked and the pay rate, and it prints the gross pay.

```
def main():
    hours = int(input("Enter hours worked: "))
    rate = float(input("Enter pay rate: "))
    pay = hours * rate
    print("Pay is $%d" % pay)

if __name__== "__main__":
  main()
```

For example,

```
Enter hours worked: 20
Enter pay rate: 25.50
Pay is $510
```

A problem arises, however, if the user types in a string rather than a number for the hours worked. For example,

```
Enter hours worked: twenty
Traceback (most recent call last):
  File "C:/Users/Garside/Documents/calculate_pay.py", line
8, in <module>
    main()
  File "C:/Users/Garside/Documents/calculate_pay.py", line
2, in main
    hours = int(input("Enter hours worked: "))
ValueError: invalid literal for int() with base 10: 'twenty'
```

The result in this case is an **exception** that produces a **Traceback**. The Traceback indicates the function and method calls that were made (most recent call last) before reaching the offending statement, which is shown as the second to last line in the Traceback output.

```
    hours = int(input("Enter hours worked: "))
```

The last line of the Traceback indicates what exception was raised. In this example, the exception is a **ValueError**, and the error includes a description of why the exception occurred.

```
ValueError: invalid literal for int() with base 10: 'twenty'
```

There are two problems with an exception like this. One, the program terminates abruptly, which could result in significant problems if, for example, a file has been opened and written to but the program is not done processing the file. Two, the Traceback that is displayed is not very friendly to a user. A program with many levels of calls can create a very long Traceback which, while useful to the programmer, is more information than the user expects to see.

In Python, we can handle exceptions like this more appropriately by writing an **exception handler** using the **try...except** statement.

Exception Handling with `try...except`

The Python **try...except** statement is used to write an **exception handler** for catching and processing semantic (runtime) errors. The general format of the statement is

```
try:
    <statements>
except: ExceptionName:
    <statements>
```

The <statements> under the **try** (also referred to as the **try block**) usually consist of one or more statements that can potentially raise an exception. If, during runtime, one of the statements raises an exception, the **except** clause is evaluated and one of the following occurs:

1. If the exception is specified by the **ExceptionName**, the <statements> immediately below the **except** clause are executed. These <statements> are referred to as the **handler** (also referred to as the **except block**). Program execution then resumes at the statement immediately following the **try...except** statement.
2. If the raised exception is **not** specified by the **ExceptionName**, the program halts and a Traceback is displayed.

Below, the pay calculator has been improved by adding a **try...except** statement.

Notice that the try block contains all of the statements in the main function. Strictly speaking, this is not necessary because only the first two statements are expected to raise an exception. However, if an exception is raised and handled, execution will then continue at the first statement following the try…except, so we don't want the last two lines of the main function to be below the try block or they would still be executed following an exception.

```
def main():
    try:
        hours = int(input("Enter hours worked: "))
        rate = float(input("Enter pay rate: "))
        pay = hours * rate
        print("Pay is $%d" % pay)
    except ValueError:
        print("error: hours worked and pay must be numbers.")

if __name__== "__main__":
  main()
```

In the code above, if either of the first two lines of the try…except block raise a ValueError, control will be transferred to the `except ValueError` handler and an error message will be printed. For example,

```
Enter hours worked: twenty
error: hours worked and pay must be numbers
```

Multiple exception types.

Consider the following modification to the pay calculator that reads the pay rate from a file called **pay_rate.txt**.

```
def main():
    try:
        hours = int(input("Enter hours worked: "))
        f = open('pay_rate.txt', 'r')  # Open file payrate.txt
        rate = float(f.readline())     # Read the pay rate.
        f.close()                      # Close the file
        pay = hours * rate
        print("Pay is $%d" % pay)
    except ValueError:
        print("error: hours worked and pay must be numbers.")

if __name__== "__main__":
  main()
```

If the file **pay_rate.txt** does not exist, the **open** statement will raise an exception. Even though we have an exception handler, notice what happens.

```
Enter hours worked: 20
Traceback (most recent call last):
  File "C:/Users/Garside/Documents/calculate_pay.py", line 22, in
<module>
    main()
  File "C:/Users/Garside/Documents/calculate_pay.py", line 13, in
main
    f = open('pay_rate.txt', 'r')
FileNotFoundError: [Errno 2] No such file or directory:
'pay_rate.txt'
```

The problem is that the **open** statement is raising a **FileNotFoundError**, rather than a **ValueError**. To handle this we can add a second exception handler as follows:

```
def main():
    try:
        hours = int(input("Enter hours worked: "))
        f = open('pay_rate.txt', 'r')  # Open file payrate.txt
        rate = float(f.readline())     # Read the pay rate.
        f.close()                      # Close the file
        pay = hours * rate
        print("Pay is $%d" % pay)
    except ValueError:
        print("error: hours worked and pay must be numbers.")
    except FileNotFoundError:
        print("error: the file \'pay_rate.txt\' was not found.")

if __name__== "__main__":
  main()
```

When this updated program is run, if the file **pay_rate.txt** does not exist, the following error will be displayed.

```
Enter hours worked: 20
error: the file 'pay_rate.txt' was not found.
```

A Catch-All Except Clause.

When you are creating a program it's easy to miss a case where an exception might occur, or more likely, miss the **type** of exception that will be raised. In order to handle this you can include a catch-all except statement that doesn't specify a specific ExceptionName. Such an exception will catch any exception that is not caught by one of the earlier **except** clauses. The following example of the pay calculator program includes a catch-all except statement that prints a generic error. While a programmer may wish to see a Traceback in this case, a simple error message is preferable for the user.

```
def main():
    try:
        hours = int(input("Enter hours worked: "))
        f = open('pay_rate.txt', 'r')  # Open file payrate.txt
        rate = float(f.readline())     # Read the pay rate.
        f.close()                      # Close the file
        pay = hours * rate
        print("Pay is $%d" % pay)
    except ValueError:
        print("error: hours worked and pay must be numbers.")
    except FileNotFoundError:
        print("error: the file \'pay_rate.txt\' was not found.")
    except:
        print("error: an error occurred during processing")

if __name__== "__main__":
  main()
```

The `else` Clause.

Earlier we indicated that all of the statements of the pay calculator were placed within the **try block** so that the last two statements, the ones that calculate and print the pay, would not be executed if an exception occurred. A better solution to this problem is to use an **else** clause.

At the end of a **try...except** statement, after all the **except clauses**, you can add an **else clause**, followed by a collection of statements. The statements

within the **else clause** are executed **only** if an exception is **not** raised. The general format of the **try…except** statement becomes:

```
try:
   <statements>
except: ExceptionName:
   <statements>
else:
   <statements>
```

The following updated pay calculator includes an **else clause.** The two statements that calculate and print the pay have been moved to the **else clause** so that they are only executed if no exception occurs.

```
def main():
    try:
        hours = int(input("Enter hours worked: "))
        f = open('pay_rate.txt', 'r')  # Open file payrate.txt
        rate = float(f.readline())     # Read the pay rate.
        f.close()                      # Close the file
    except ValueError:
        print("error: hours worked and pay must be numbers.")
    except FileNotFoundError:
        print("error: the file \'pay_rate.txt\' was not found.")
    except:
        print("error: an error occurred during processing")
    else:
        pay = hours * rate
        print("Pay is $%d" % pay)


if __name__== "__main__":
  main()
```

Exiting a program on error.

While the pay calculator program is functional, it is simple. More complicated programs are likely to have multiple functions and even multiple modules. In programs like these, it's is often desirable to exit the program when an exception occurs so that subsequent statements or functions are not executed.

If it is desired to exit the program when an exception is handled, the preferred way to do it is using the **exit()** function from the **sys** module. The **sys** module contains numerous functions and variables for manipulating various parts of the Python runtime environment. The **sys** module's **exit()** function terminates program execution and returns a numeric status specified by the first argument. When a program exits successfully, the return value should be 0. When a program exits due to an exception, the return value should be non-zero, typically, 1.

The final example of the pay calculator appears below. Notice it **imports** the **sys** module and that in each **except clause** the last statement is **sys.exit(1)**. When executed, the **sys.exit(1)** statement will cause the program to exit immediately and return a program completion status of 1.

```
import sys

def main():
    try:
        hours = int(input("Enter hours worked: "))
        f = open('pay_rate.txt', 'r')  # Open file payrate.txt
        rate = float(f.readline())     # Read the pay rate.
        f.close()                      # Close the file
    except ValueError:
        print("error: hours worked and pay must be numbers.")
        sys.exit(1)
    except FileNotFoundError:
        print("error: the file \'pay_rate.txt\' was not found.")
        sys.exit(1)
    except:
        print("error: an error occurred during processing")
        sys.exit(1)
    else:
        pay = hours * rate
        print("Pay is $%d" % pay)
```

```
if __name__== "__main__":
  main()
```

Object-based Programing using simple Graphics

Object-based programming is a form of programming that relies on objects and methods to control the complexity of a program. In this section we will use a simple graphics module called **turtle** to explore classes, objects and methods. The Turtle graphics toolkit provides a simple way to draw in a window and provides the opportunity to examine an object (the **turtle** object) and the manner in which multiple methods of the object may be applied.

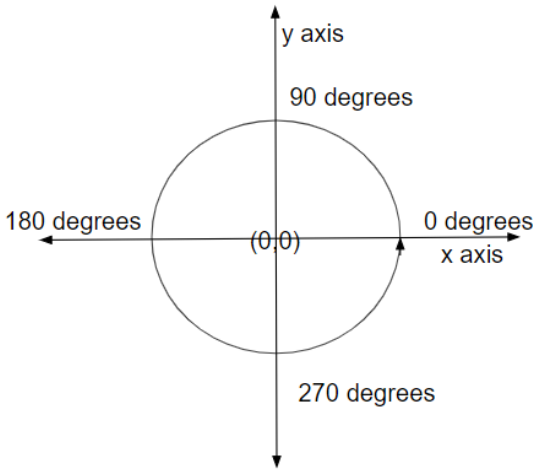
Overview

As we have seen throughout our lectures, Python uses **objects** to represent “things” like, strings, integers, lists, tuples, and dictionaries, and each of these objects has a well-defined set of **methods** for manipulating the object. In **object-oriented programming (OOP)**, the code that defines each object is known as a **class**, and when an **instance** of the class is created (**instantiated**), it is known as an **object**. As we look more closely at **object-based programming** we will see that every day “things”, from people to automotive parts, can be represented as objects, with a set of clearly defined methods to operate on each type of object.

In Turtle graphics, the drawing object is known as a **turtle** (an analogy to a turtle whose tail is used to draw lines) and operations on the turtle object are defined as methods.

The coordinate system for Turtle graphics is a standard Cartesian Coordinate system. The origin, (0,0), is located at the center of the window; the initial drawing position (also referred to as **home**) is at this origin; and the initial drawing direction (**heading**) is 0 degrees.

The following diagram describes the coordinate system.



In **object-based programming** an object is considered to have a **state**. Basically, an object's **state** is the set of values of its attributes at any given point in time. With the turtle object, the **state** of the object is given by the following attributes.

heading

The current direction of drawing, specified in degrees. As degrees **increase**, the direction moves counterclockwise around the circle. As degrees **decrease**, the direction moves clockwise.

Color

The color used when drawing. The initial color is black.

Width

The width (pen size) of the line drawn, in pixels.

Down

Indicates whether the pen (the turtle's tail) is down. When **True**, a line is drawn as the drawing position moves. When **False**, the position is changed without drawing.

The interface

The set of **methods** of a given class of objects is called its **interface**. Ideally, the interface should contain only the information necessary to use (manipulate) the object, including the method headers, documentation about the method arguments, values returned, and changes to the state of the object.

To import the Turtle **interface** you use the following command

```
from turtle import Turtle
```

As we have seen in earlier lectures, you can access a module's doc string from the Python shell using the **help()** function. Using the syntax **help(<class-name>)** you can access the interface (doc string) of a given class, or you can use **help(<class-name>.<method>)** to access the doc string of a particular class method.

For example, once imported, you can view the interface of Turtle's **width()** method using the following command.

```
>>> help(Turtle.width)
Help on function pensize in module turtle:

pensize(self, width=None)
    Set or return the line thickness.

    Aliases:  pensize | width

    Argument:

    width -- positive number

    Set the line thickness to width or return it. If resizemode is set
    to "auto" and turtleshape is a polygon, that polygon is drawn with
    the same line thickness. If no argument is given, current pensize
    is returned.

    Example (for a Turtle instance named turtle):
    >>> turtle.pensize()
    1
    >>> turtle.pensize(10)   # from here on lines of width 10 are drawn
```

As we have also seen in past lectures, you can get the complete list of a class' variables and methods using the **dir** function. Below, we do this for the Turtle class.

```
>>> dir(Turtle)
['DEFAULT_ANGLEOFFSET', 'DEFAULT_ANGLEORIENT', 'DEFAULT_MODE',
'START_ORIENTATION', '__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_cc', '_clear',
'_clearstamp', '_color', '_colorstr', '_delay', '_drawturtle', '_getshapepoly', '_go',
'_goto', '_newLine', '_pen', '_polytrafo', '_reset', '_rotate', '_screen',
'_setDegreesPerAU', '_setmode', '_tracer', '_undo', '_undogoto', '_update',
'_update_data', '_write', 'back', 'backward', 'begin_fill', 'begin_poly', 'bk', 'circle',
'clear', 'clearstamp', 'clearstamps', 'clone', 'color', 'degrees', 'distance', 'dot',
'down', 'end_fill', 'end_poly', 'fd', 'fillcolor', 'filling', 'forward', 'get_poly',
'get_shapepoly', 'getpen', 'getscreen', 'getturtle', 'goto', 'heading', 'hideturtle',
'home', 'ht', 'isdown', 'isvisible', 'left', 'lt', 'onclick', 'ondrag', 'onrelease', 'pd',
'pen', 'pencolor', 'pendown', 'pensize', 'penup', 'pos', 'position', 'pu', 'radians',
'reset', 'resizemode', 'right', 'rt', 'screens', 'seth', 'setheading', 'setpos',
'setposition', 'settiltangle', 'setundobuffer', 'setx', 'sety', 'shape', 'shapesize',
'shapetransform', 'shearfactor', 'showturtle', 'speed', 'st', 'stamp', 'tilt', 'tiltangle',
'towards', 'turtlesize', 'undo', 'undobufferentries', 'up', 'width', 'write', 'xcor',
'ycor']
>>>
```

The following table describes in more detail some of the methods shown above. In the table, **t** represents a turtle object.

Method	Description
t.home()	Moves **t** to the center of the window and points **t** along the x axis (0 degrees)
t.up()	Raise **t**'s pen from the drawing surface.
t.down()	Lower **t**'s pen to the drawing surface.
t.setheading(degrees)	Point **t** in the indicated direction, specified in degrees.
t.left(degrees)	Rotate **t** to the left (counter clockwise) by the specified degrees.
t.right(degrees)	Rotate **t** to the right (clockwise) by the specified degrees.

t.goto(x, y)	Moves **t** to the specified position on the Cartesian Coordinates.
t.forward(distance)	Moves **t** the specified distance in the current direction.
t.pencolor(r, g, b) **t.pencolor(string)**	Change the pen color of **t** to the specified red, green, blue (RGB) value, or to the specified string, such as "**red**". When arguments are omitted, it returns the current pen color value.
t.fillcolor(r, g, b) **t.fillcolor(string)**	Change the fill color of **t** to the specified red, green, blue (RGB) value, or to the specified string, such as "**red**". When arguments are omitted, it returns the current fill color value.
t.begin_fill() **t.end_fill()**	Enclose a set of turtle commands that will draw a filled shape using the current fill color.
t.clear()	Erases all of the turtle's drawings, without changing the turtle's state
t.width(pixels)	Changes the width of **t** to the specified number of pixels. Returns **t**'s current width when the argument is omitted.
t.hideturtle() **t.showturtle()**	Makes the turtle (pen) invisible or visible.
t.position()	Returns the current position (x, y) of **t**.
t.heading()	Returns the current direction of t.
t.isdown()	Returns **True** if t's pen is down. Otherwise, **False**.

Object Instantiation

Before you can use an object it must be created, or, to be more precise, an **instance** of the object's class must be created. The process of creating an object is called **instantiation**.

For all of the objects we have seen so far, Python has instantiated instances of those objects (from their respective classes) whenever we have need them. For example, whenever an assignment of the following form occurs, Python instantiates a new string (str) object to hold the text string.

```
saying = "Hello World!"
```

A similar thing is done for integers, floats, lists, tuples and dictionaries.

For other types of objects, including Turtle objects, and any other objects instantiated from classes a programmer typically creates, you must explicitly request the object to be instantiated (created).

The syntax for instantiating a class and assigning the resulting object to a variable is as follows:

<variable-name> = <class-name>(<arguments>)

The expression on the right side of the assignment resembles a function call and is called a **constructor**. The **constructor** is responsible for creating and returning a new instance of the class.

The arguments, if any, provide initial values for the new object's attributes, or other information needed to create the object. With constructors, it is common to have optional arguments, which are given reasonable values if the associated arguments are not provided.

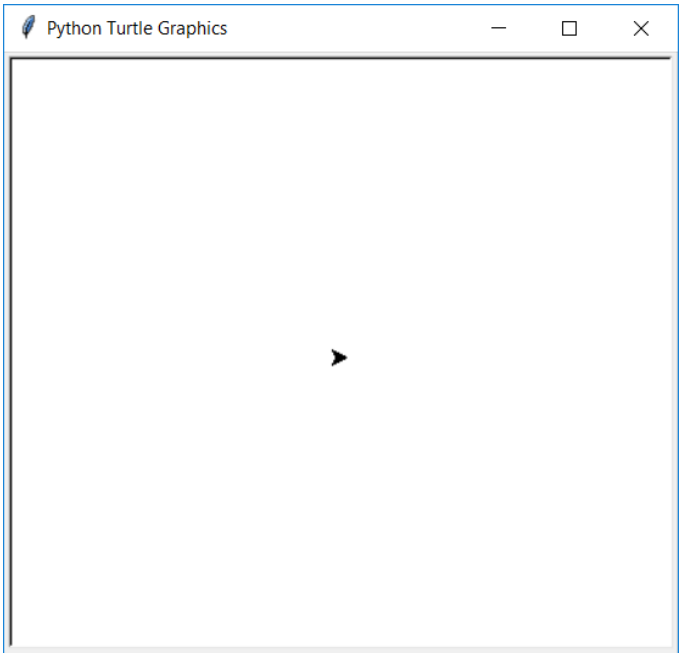
The Turtle class is defined in the turtle module. To instantiate an instance of the Turtle class, we first import the Turtle **interface** using the following command:

```
from turtle import Turtle
```

We can then **instantiate** an **instance** of the Turtle **class** (that is, create a Turtle **object**) using a command like the following:

```
t = Turtle()
```

If you enter the two statements above into a Python program and run it, or enter them into the Python Shell, a Turtle Graphics window like the following will appear with the drawing arrow/pen (i.e. the Turtle) displayed in the center of the window at coordinates (0,0) and facing in the direction of 0 degrees.



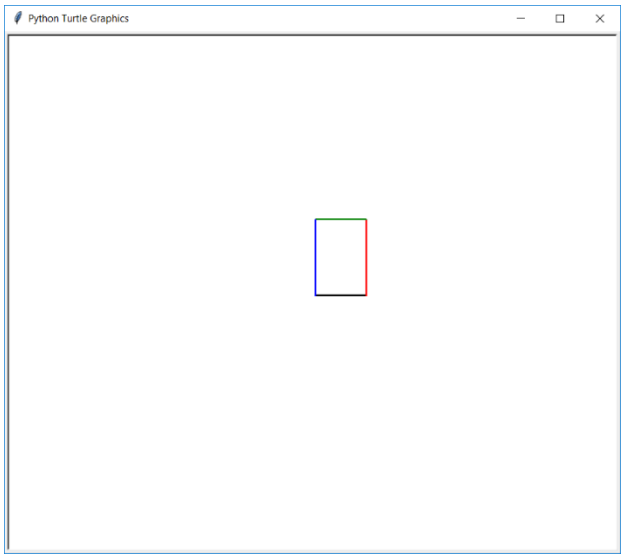
Using methods to draw

Using the Turtle object methods we saw earlier, we can now draw within the window.

The following code imports the Turtle interface from the turtle module, instantiates an instance of the Turtle class (creates a Turtle object), and uses a number of the methods we defined earlier to draw a multicolored rectangle.

```
from turtle import Turtle

t = Turtle()        # Instantiate an instance of the Turtle class.

                    # Draw a multi-colored rectangle.

t.width(2)          # Make the lines two pixels wide.
t.forward(60)       # Draw a line 60 pixes long. Since the default direction.
                    # is 0 degress, the line is drawn along the x-axis.
t.left(90)          # Change the direction 90 degrees to the left.
                    # This makes the direction straight up along the y-axis.
t.pencolor("red")   # Let's draw in red now.
t.forward(90)       # Draw a line 60 pixes long.

                    # Draw a green line across the top of the rectangle.
t.left(90)
t.pencolor("green")
t.forward(60)

                     # Draw the final line of the rectangle in green.
t.left(90)
t.pencolor("Blue")
t.forward(90)

t.hideturtle()      # Now that we're done drawing, hide the pen (Turtle)
```

When the program is run, the window below appears along with a multicolored rectangle.



A complete example

We can now pull a number of the concepts we've learned together into a complete program.

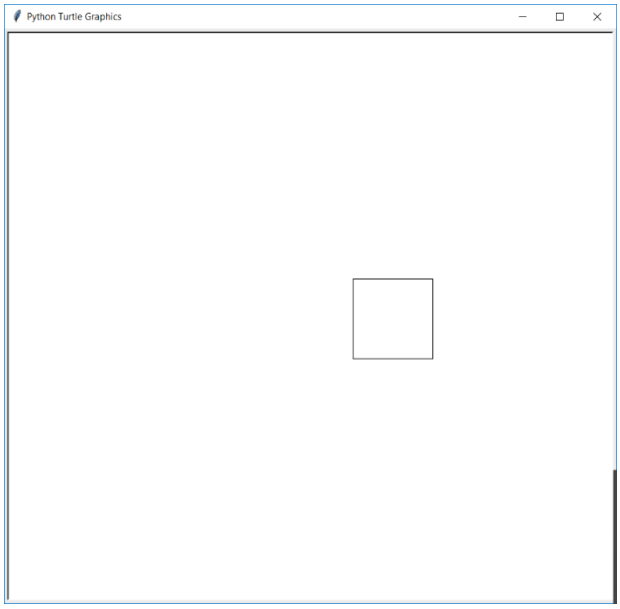
The following program includes a function called **draw_square** which takes a Turtle object, **t**, and draws a square with sides of length, **length**, with the upper-left corner of the square at coordinates (x,y).

The main program creates a window by **instantiating** an **instance** of the Turtle **class** (a Turtle **object**), **t**, and then calls **draw_square** to draw a square with side lengths of 100 pixels and the upper-left corner of the square at point (50,50). The last line of the main function hides the pen so the view of the square is not occluded by the pen.

```
from turtle import Turtle


def draw_square(t, x, y, length):
    """Draws a square at the upper-left corner point (x,y) with
       sides of "length" for the given Turtle object, t.
    """
    t.up()                  # Raise the pen from the drawing surface.
    t.goto(x,y)             # Move to point x,y
    t.setheading(270)       # Set the heading straight down along the y-axis.
    t.down()                # Lower the pen for writing.
    for count in range(4):  # Draw all four sides of the square.
        t.forward(length)
        t.left(90)
```

```
if __name__ == "__main__":
    t = Turtle()                # Instantiate an instance of the Turtle class.
    draw_square(t,50,50,100)    # Draw a square with sides 100 pixels in length
                                # with the upper-left corner at poing (50,50)
    t.hideturtle()              # Hide the cursor (pen)
```

When the program is run, the following window and drawing are displayed.



Examining an Object's Attributes

The methods used in the above examples modify a Turtle object's attributes, such as position, heading and color. Such methods are called **mutator methods**, meaning they change the internal state of the object.

A number of the other Turtle methods simply return the values of a Turtle object's attributes without altering the object's state. Such methods are called **accessor methods**. Some of Turtle's **accessor methods** include **position**, **heading** and **isdown**.

The following Python Shell session instantiates a Turtle object and displays the value of a number of the object's attributes.

```
>>> from turtle import Turtle
>>> t = Turtle()
>>> t.position()
(0.00,0.00)
>>> t.heading()
0.0
>>> t.isdown()
True
>>>
```

There are many things that can be done with Turtle. I encourage you to explore the listed methods and apply them to create new and different outputs.

Conclusion

In this lecture we have expanded our understanding of Python Variables (module variables, local/temporary variables and function parameters) along with their scopes and lifetimes. We have also extended our understanding of Python functions, including optional arguments, anonymous (lambda) functions, and passing functions as arguments to functions using higher-order functions like **map**(), **filter**() and **reduce**(). Finally, we looked at exception handling in Python and took a closer look at object-based programming concepts, including classes, objects and methods while investigating simple graphics.