MODULE-4

MODULES, MUTABLE VERSUS IMMUTABLE & ALIASING AND OBJECT-ORIENTED PROGRAMMING

SYLLABUS:
Modules: *Random numbers, the time module, the math module, creating your own modules, Namespaces, Scope and lookup rules, Attributes and the dot Operator, Three import statement variants.*
Mutable versus immutable and aliasing
Object oriented programming: *Classes and Objects — The Basics, Attributes, Adding methods to our class, Instances as arguments and parameters, Converting an instance to a string, Instances as return values.*

CHAPTERS: *8.1-8.8, 9.1, 11.1*

TEXT BOOK: *Peter Wentworth, Jeffrey Elkner, Allen B. Downey and Chris Meyers- How to think like a computer scientist: learning with python 3. Green Tea Press, Wellesley, Massachusetts, 2020*

Lecture - 25

MODULES

- A module is a file containing Python definitions and statements intended for use in other Python programs.
- There are many Python modules that come with Python as part of the standard library.

RANDOM NUMBERS

- We often want to use random numbers in programs, here are a few typical uses:
  - To play a game of chance where the computer needs to throw some dice, pick a number, or flip a coin,
  - To shuffle a deck of playing cards randomly,
  - To allow/make an enemy spaceship appear at a random location and start shooting at the player,
  - To simulate possible rainfall when we make a computerized model for estimating the environmental impact of building a dam,
  - For encrypting banking sessions on the Internet.
- Python provides a module random that helps with tasks like this. You can look it up using help, but here are the key things we'll do with it:

```
import random
# Create a black box object that generates random numbers
rng = random.Random()
dice_throw = rng.randrange(1,7) # Return an int, one of 1,2,3,4,5,6
delay_in_seconds = rng.random() * 5.0
```

- The randrange method call generates an integer between its lower and upper argument, using the same semantics as range—so the lower bound is included, but the upper bound is excluded. All the values have an equal probability of occurring (i.e. the results are uniformly distributed). Like range, randrange

can also take an optional step argument. So let's assume we needed a random odd number less than 100, we could say:

```
random_odd = rng.randrange(1, 100, 2)
```

- Other methods can also generate other distributions e.g. a bell-shaped, or "normal" distribution might be more appropriate for estimating seasonal rainfall, or the concentration of a compound in the body after taking a dose of medicine.
- The random method returns a floating point number in the interval [0.0, 1.0) — the square bracket means "closed interval on the left" and the round parenthesis means "open interval on the right". In other words, 0.0 is possible, but all returned numbers will be strictly less than 1.0. It is usual to scale the results after calling this method, to get them into an interval suitable for your application.
- In the case shown here, we've converted the result of the method call to a number in the interval [0.0, 5.0). Once more, these are uniformly distributed numbers—numbers close to 0 are just as likely to occur as numbers close to 0.5, or numbers close to 1.0.
- This example shows how to shuffle a list. (shuffle cannot work directly with a lazy promise, so notice that we had to convert the range object using the list type converter first.)

```
cards = list(range(52))          # Generate ints [0 .. 51]
                                 # representing a pack of cards.
rng.shuffle(cards)               # Shuffle the pack
```

REPEATABILITY AND TESTING

- Random number generators are based on a deterministic algorithm — repeatable and predictable. So they're called pseudo-random generators — they are not genuinely random. They start with a seed value. Each time you ask for another random number, you'll get one based on the current seed attribute, and the state of the seed (which is one of the attributes of the generator) will be updated.
- For debugging and for writing unit tests, it is convenient to have repeatability—programs that do the same thing every time they are run. We can arrange this by forcing the random number generator to be initialized with a known seed every time. (Often this is only wanted during testing — playing a game of cards where the shuffled deck was always in the same order as last time you played would get boring very rapidly!)

```
drng = random.Random(123)      # Create generator with known starting state
```

- This alternative way of creating a random number generator gives an explicit seed value to the object. Without this argument, the system probably uses something based on the time. So grabbing some random numbers from drng today will give you precisely the same random sequence as it will tomorrow!

PICKING BALLS FROM BAGS, THROWING DICE, SHUFFLING A PACK OF CARDS

- Here is an example to generate a list containing n random ints between a lower and an upper bound:

```
import random
def make_random_ints(num, lower_bound, upper_bound):
    """
        Generate a list containing num random ints between lower_bound
        and upper_bound. upper_bound is an open bound.
```

```
    """
      rng = random.Random()      # Create a random number generator
      result = []
      for i in range(num):
      result.append(rng.randrange(lower_bound, upper_bound))
  return result

>>> make_random_ints(5, 1, 13) # Pick 5 random month numbers
[8, 1, 8, 5, 6]
```

- Notice that we got a duplicate in the result. Often this is wanted, e.g. if we throw a die five times, we would expect some duplicates. But what if don't want duplicates? If wanted 5 distinct months, then this algorithm is wrong. In this case a good algorithm is to generate the list of possibilities, shuffle it, and slice off the number of elements you want:

```
xs = list(range(1,13))              # Make list 1..12 (there are no duplicates)
rng = random.Random()          # Make a random number generator
rng.shuffle(xs)                     # Shuffle the list
result = xs[:5]                     # Take the first five elements
```

- In statistics courses, the first case: allowing duplicates is usually described as pulling balls out of a bag with replacement, put the drawn ball back in each time, so it can occur again. The latter case, with no duplicates, is usually described as pulling balls out of the bag without replacement. Once the ball is drawn, it doesn't go back to be drawn again. TV lotto games work like this.
- The second "shuffle and slice" algorithm would not be so great if you only wanted a few elements, but from a very large domain. Suppose I wanted five numbers between one and ten million, without duplicates. Generating a list of ten million items, shuffling it, and then slicing off the first five would be a performance disaster! So let us have another try:

```
import random
def make_random_ints_no_dups(num, lower_bound, upper_bound):
    """
      Generate a list containing num random ints between
      lower_bound and upper_bound. upper_bound is an open bound.
      The result list cannot contain duplicates.
    """
    result = []
    rng = random.Random()
    for i in range(num):
        while True:
            candidate = rng.randrange(lower_bound, upper_bound)
            if candidate not in result:
                break
        result.append(candidate)
    return result
xs = make_random_ints_no_dups(5, 1, 10000000)
print(xs)
```

This agreeably produces 5 random numbers, without duplicates:

[3344629, 1735163, 9433892, 1081511, 4923270]

- Even this function has its pitfalls. Can you spot what is going to happen in this case?

```
xs = make_random_ints_no_dups(10, 1, 6)
```

Review questions:

1. What is module?
2. What is random?
3. Name two common use of random numbers in programming.
4. What does the randrange method do?

Lecture - 26

THE TIME MODULE

- As we start to work with more sophisticated algorithms and bigger programs, a natural concern is "is our code efficient?" One way to experiment is to time how long various operations take. The time module has a function called clock that is recommended for this purpose. Whenever clock is called, it returns a floating point number representing how many seconds have elapsed since your program started running.
- The way to use it is to call clock and assign the result to a variable, say t0, just before you start executing the code you want to measure. Then after execution, call clock again, (this time we'll save the result in variable t1). The difference t1-t0 is the time elapsed, and is a measure of how fast your program is running.
- Let's try a small example. Python has a built-in sum function that can sum the elements in a list. We can also write our own. How do we think they would compare for speed? We'll try to do the summation of a list [0, 1, 2 . . .] in both cases, and compare the results:

```
import time
def do_my_sum(xs):
    sum = 0
    for v in xs:
        sum += v
    return sum

sz = 10000000              # Lets have 10 million elements in the list
testdata = range(sz)

t0 = time.clock()
my_result = do_my_sum(testdata)
t1 = time.clock()
print("my_result = {0} (time taken = {1:.4f} seconds)"
        .format(my_result, t1-t0))

t2 = time.clock()
their_result = sum(testdata)
t3 = time.clock()
print("their_result = {0} (time taken = {1:.4f} seconds)"
    .format(their_result, t3-t2))
```

On a reasonably modest laptop, we get these results:

```
my_sum = 49999995000000 (time taken = 1.5567 seconds)
their_sum = 49999995000000 (time taken = 0.9897 seconds)
```

- So our function runs about 57% slower than the built-in one. Generating and summing up ten million elements in under a second is not too shabby!

THE MATH MODULE

- The math module contains the kinds of mathematical functions typically found on calculator (sin, cos, sqrt, asin, log, log10) and some mathematical constants like pi and e:

```
>>> import math
>>> math.pi                          # Constant pi
3.141592653589793
>>> math.e                           # Constant natural log base
2.718281828459045
>>> math.sqrt(2.0)                   # Square root function
1.4142135623730951
>>> math.radians(90)                 # Convert 90 degrees to radians
1.5707963267948966
>>> math.sin(math.radians(90))       # Find sin of 90 degrees
1.0
>>> math.asin(1.0) * 2               # Double the arcsin of 1.0 to get pi
3.141592653589793
```

- Like almost all other programming languages, angles are expressed in radians rather than degrees. There are two functions radians and degrees to convert between these two popular ways of measuring angles.
- Notice another difference between this module and our use of random and turtle: in random and turtle we create objects and we call methods on the object. This is because objects have state, a turtle has a color, a position, a heading, etc., and every random number generator has a seed value that determines its next result.
- Mathematical functions are "pure" and don't have any state, calculating the square root of 2.0 doesn't depend on any kind of state or history about what happened in the past. So, the functions are not methods of an object; they are simply functions that are grouped together in a module called math.

CREATING YOUR OWN MODULES

- All we need to do to create our own modules is to save our script as a file with a .py extension. Suppose, for example, this script is saved as a file named seqtools.py:

```
def remove_at(pos, seq):
    return seq[:pos] + seq[pos+1:]
```

- We can now use our module, both in scripts we write, or in the interactive Python interpreter. To do so, we must first import the module.

```
>>> import seqtools
>>> s = "A string!"
>>> seqtools.remove_at(4, s)
```

```
'A sting!'
```

- We do not include the .py file extension when importing. Python expects the file names of Python modules to end in .py, so the file extension is not included in the import statement.
- The use of modules makes it possible to break up very large programs into manageable sized parts, and to keep related parts together.

Review questions:

1. What is the purpose of the time module in Python?
2. What is the purpose of the math module in Python?
3. What does the time module allow you to measure in a Python program?
4. What file extension must a Python module have?

Lecture - 27

NAMESPACES

- A namespace is a collection of identifiers that belong to a module, or to a function, (and as we will see soon, in classes too). Generally, we like a namespace to hold "related" things, e.g. all the math functions, or all the typical things we'd do with random numbers.
- Each module has its own namespace, so we can use the same identifier name in multiple modules without causing an identification problem.

```
# module1.py
question = "What is the meaning of Life, the Universe, and Everything?"
answer = 42

# module2.py
question = "What is your quest?"
answer = "To seek the holy grail."
```

- We can now import both modules and access question and answer in each:

```
import module1
import module2

print(module1.question)
print(module2.question)
print(module1.answer)
print(module2.answer)
```

will output the following:

```
What is the meaning of Life, the Universe, and Everything?
What is your quest?
42
To seek the holy grail.
```

- Functions also have their own namespaces:

```
def f():
    n = 7
```

```
    print("printing n inside of f:", n)
def g():
    n = 42
    print("printing n inside of g:", n)
n = 11
print("printing n before calling f:", n)
f()
print("printing n after calling f:", n)
g()
print("printing n after calling g:", n)
```

Running this program produces the following output:

```
printing n before calling f: 11
printing n inside of f: 7
printing n after calling f: 11
printing n inside of g: 42
printing n after calling g: 11
```

- The three n's here do not collide since they are each in a different namespace—they are three names for three different variables, just like there might be three different instances of people, all called "Bruce".
- Namespaces permit several programmers to work on the same project without having naming collisions.

HOW ARE NAMESPACES, FILES AND MODULES RELATED?

- Python has a convenient and simplifying one-to-one mapping, one module per file, giving rise to one namespace. Also, Python takes the module name from the file name, and this becomes the name of the namespace. math.py is a filename, the module is called math, and its namespace is math. So, in Python the concepts are more or less interchangeable.
- But will encounter other languages (e.g. C#), that allow one module to span multiple files, or one file to have multiple namespaces, or many files to all share the same namespace. So, the name of the file doesn't need to be the same as the namespace.
- Files and directories organize where things are stored in our computer. On the other hand, namespaces and modules are a programming concept: they help us organize how we want to group related functions and attributes. They are not about "where" to store things, and should not have to coincide with the file and directory structures.
- So, in Python, if you rename the file math.py, its module name also changes, your import statements would need to change, and code that refers to functions or attributes inside that namespace would also need to change.

SCOPE AND LOOKUP RULES

- The scope of an identifier is the region of program code in which the identifier can be accessed, or used.
- There are three important scopes in Python:
  - Local scope refers to identifiers declared within a function. These identifiers are kept in the namespace that belongs to the function, and each function has its own namespace.
  - Global scope refers to all the identifiers declared within the current module, or file.
  - Built-in scope refers to all the identifiers built into Python — those like range and min that can be used without having to import anything, and are (almost) always available.

- Python (like most other computer languages) uses precedence rules: the same name could occur in more than one of these scopes, but the innermost, or local scope, will always take precedence over the global scope, and the global scope always gets used in preference to the built-in scope. Let's start with a simple example:

```
def range(n):
    return 123*n
print(range(10))
```

- What gets printed? We've defined our own function called range, so there is now a potential ambiguity. When we use range, do we mean our own one, or the built-in one? Using the scope lookup rules determines this: our own range function, not the built-in one, is called, because our function range is in the global namespace, which takes precedence over the built-in names.
- So although names likes range and min are built-in, they can be "hidden" from your use if you choose to define your own variables or functions that reuse those names. Now, a slightly more complex example:

```
n = 10
m = 3
def f(n):
    m = 7
    return 2*n+m
print(f(5), n, m)
```

- This prints 17 10 3. The reason is that the two variables m and n in lines 1 and 2 are outside the function in the global namespace. Inside the function, new variables called n and m are created just for the duration of the execution of f. These are created in the local namespace of function f.
- Within the body of f, the scope lookup rules determine that we use the local variables m and n. By contrast, after we've returned from f, the n and m arguments to the print function refer to the original variables on lines 1 and 2, and these have not been changed in any way by executing function f.
- Notice too that the def puts name f into the global namespace here. So, it can be called on line 7. What is the scope of the variable n on line 1? Its scope — the region in which it is visible — is lines 1, 2, 6, 7. It is hidden from view in lines 3, 4, 5 because of the local variable n.

Review questions:

1. What is a namespace in Python?
2. What types of namespaces exist in Python?
3. What does local scope and global scope refer to?
4. What is the relationship between files, modules, and namespaces in Python?

Lecture - 28

ATTRIBUTES AND THE DOT OPERATOR

- Variables defined inside a module are called attributes of the module. We've seen that objects have attributes too: for example, most objects have a __doc__ attribute, some functions have a __annotations__ attribute. Attributes are accessed using the dot operator (.). The question attribute of module1 and module2 is accessed using module1.question and module2.question.
- Modules contain functions as well as attributes, and the dot operator is used to access them in the same way. seqtools.remove_at refers to the remove_at function in the seqtools module.

- When we use a dotted name, we often refer to it as a fully qualified name, because we're saying exactly which question attribute we mean.

THREE IMPORT STATEMENT VARIANTS

- Here are three different ways to import names into the current namespace, and to use them:

```
import math
x = math.sqrt(10)
```

- Here just the single identifier math is added to the current namespace. If you want to access one of the functions in the module, you need to use the dot notation to get to it. Here is a different arrangement:

```
from math import cos, sin, sqrt
x = sqrt(10)
```

- The names are added directly to the current namespace, and can be used without qualification. The name math is not itself imported, so trying to use the qualified form math.sqrt would give an error. Then we have a convenient shorthand:

```
from math import *          # Import all the identifiers from math,
                            # adding them to the current namespace.
x = sqrt(10)                # Use them without qualification.
```

- Of these three, the first method is generally preferred, even though it means a little more typing each time. Although, we can make things shorter by importing a module under a different name:

```
>>> import math as m
>>> m.pi
3.141592653589793
```

But with nice editors that do auto-completion, and fast fingers, that's a small price! Finally, observe this case:

```
def area(radius):
    import math
    return math.pi * radius * radius
x = math.sqrt(10)           # This gives an error
```

- Here we imported math, but we imported it into the local namespace of area. So, the name is usable within the function body, but not in the enclosing script, because it is not in the global namespace.

Review questions:

1. What is the use of dot operator?
2. What are three different ways to import names from a module into the current Python namespace?
3. What are variables defined inside a module called?
4. What is the difference between import math and from math import sqrt?

Lecture - 29

MUTABLE VERSUS IMMUTABLE AND ALIASING

- Some datatypes in Python are mutable. This means their contents can be changed after they have been created. Lists and dictionaries are good examples of mutable datatypes.

```
>>> my_list = [2, 4, 5, 3, 6, 1]
>>> my_list[0] = 9
>>> my_list
[9, 4, 5, 3, 6, 1]
```

- Tuples and strings are examples of immutable datatypes, their contents can not be changed after they have been created:

```
>>> my_tuple = (2, 5, 3, 1)
>>> my_tuple[0] = 9
Traceback (most recent call last):
  File "<interactive input>", line 2, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

- Mutability is usually useful, but it may lead to something called aliasing. In this case, two variables refer to the same object and mutating one will also change the other:

```
>>> list_one = [1, 2, 3, 4, 6]
>>> list_two = list_one
>>> list_two[-1] = 5
>>> list_one
[1, 2, 3, 4, 5]
```

- This happens, because both list_one and list_two refer to the same memory address containing the actual list. Check this using the built-in function id:

```
>>> list_one = [1, 2, 3, 4, 6]
>>> list_two = list_one
>>> id(list_one) == id(list_two)
True
```

- We can escape this problem by making a copy of the list:

```
>>> list_one = [1, 2, 3, 4, 6]
>>> list_two = list_one[:]
>>> id(list_one) == id(list_two)
False
>>> list_two[-1] = 5
>>> list_two
[1, 2, 3, 4, 5]
>>> list_one
[1, 2, 3, 4, 6]
```

- However, this will not work for nested lists because of the same reason. The module copy provides functions to solve this.

Review questions:

1. What is the difference between mutable and immutable datatypes in Python?
2. Give two examples of mutable datatypes and two examples of immutable datatypes.
3. What happens when two variables refer to the same mutable object?
4. How can changing one variable affect another when aliasing occurs?

Lecture - 30

CLASSES AND OBJECTS: THE BASICS

- Python is an object-oriented programming language, which means that it provides features that support object-oriented programming (OOP).
- Object-oriented programming has its roots in the 1960s, but it wasn't until the mid 1980s that it became the main programming paradigm used in the creation of new software. It was developed as a way to handle the rapidly increasing size and complexity of software systems, and to make it easier to modify these large and complex systems over time.
- Up to now, most of the programs we have been writing use a procedural programming paradigm. In procedural programming the focus is on writing functions or procedures which operate on data. In object-oriented programming the focus is on the creation of objects which contain both data and functionality together. (We have seen turtle objects, string objects, and random number generators, to name a few places where we've already worked with objects.)
- Usually, each object definition corresponds to some object or concept in the real world, and the functions that operate on that object correspond to the ways real-world objects interact.

USER-DEFINED COMPOUND DATA TYPES

- Consider the concept of a mathematical point. In two dimensions, a point is two numbers (coordinates) that are treated collectively as a single object. Points are often written in between parentheses with a comma separating the coordinates. For example, (0, 0) represents the origin, and (x, y) represents the point x units to the right and y units up from the origin.
- Some of the typical operations that one associates with points might be calculating the distance of a point from the origin, or from another point, or finding a midpoint of two points, or asking if a point falls within a given rectangle or circle.
- A natural way to represent a point in Python is with two numeric values. The question, then, is how to group these two values into a compound object. The quick and dirty solution is to use a tuple, and for some applications that might be a good choice.
- An alternative is to define a new class. This approach involves a bit more effort, but it has advantages that will be apparent soon. We'll want our points to each have an x and a y attribute, so our first-class definition looks like this:

```
class Point:
    """ Point class represents and manipulates x,y coords. """
    def __init__(self):
        """ Create a new point at the origin """
        self.x = 0
        self.y = 0
```

- Class definitions can appear anywhere in a program, but they are usually near the beginning (after the import statements). Some programmers and languages prefer to put every class in a module of its own — we won't do that here.
- The syntax rules for a class definition are the same as for other compound statements. There is a header which begins with the keyword, class, followed by the name of the class, and ending with a colon. Indentation levels tell us where the class ends.
- If the first line after the class header is a string, it becomes the docstring of the class, and will be recognized by various tools. (This is also the way docstrings work in functions.)
- Every class should have a method with the special name __init__. This initializer method is automatically called whenever a new instance of Point is created. It gives the programmer the opportunity to set up the attributes required within the new instance by giving them their initial state/values.
- The self-parameter (we could choose any other name, but self is the convention) is automatically set to reference the newly created object that needs to be initialized. So, let's use our new Point class now:

```
p = Point()                    # Instantiate an object of type Point
q = Point()                    # Make a second point
print(p.x, p.y, q.x, q.y)      # Each point object has its own x and y
```

This program prints:

```
0 0 0 0
```

- Because during the initialization of the objects, we created two attributes called x and y for each, and gave them both the value 0. This should look familiar— we've used classes before to create more than one object:

```
from turtle import Turtle
tess = Turtle()                # Instantiate objects of type Turtle
alex = Turtle()
```

- The variables p and q are assigned references to two new Point objects. A function like Turtle or Point that creates a new object instance is called a constructor, and every class automatically provides a constructor function which is named the same as the class.
- It may be helpful to think of a class as a factory for making objects. The class itself isn't an instance of a point, but it contains the machinery to make point instances. Every time we call the constructor, we're asking the factory to make us a new object. As the object comes off the production line, its initialization method is executed to get the object properly set up with its factory default settings.
- The combined process of "make me a new object" and "get its settings initialized to the factory default settings" is called instantiation.

Review questions:

1. What is object-oriented programming (OOP)?
2. What does a class definition represent in Python?
3. What is object?
4. What is the purpose of the __init__ method?

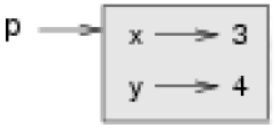
Lecture - 31

ATTRIBUTES

- Object instances have both attributes and methods. We can modify the attributes in an instance using dot notation:

```
>>> p.x = 3
>>> p.y = 4
```

- Both modules and instances create their own namespaces, and the syntax for accessing names contained in each, called attributes, is the same. In this case the attribute we are selecting is a data item from an instance. The following state diagram shows the result of these assignments:



- The variable p refers to a Point object, which contains two attributes. Each attribute refers to a number. We can access the value of an attribute using the same syntax:

```
>>> print(p.y)
4
>>> x = p.x
>>> print(x)
3
```

- The expression p.x means, “Go to the object p refers to and get the value of x”. In this case, we assign that value to a variable named x. There is no conflict between the variable x (in the global namespace here) and the attribute x (in the namespace belonging to the instance). The purpose of dot notation is to fully qualify which variable we are referring to unambiguously.
- We can use dot notation as part of any expression, so the following statements are legal:

```
print("(x={0}, y={1})".format(p.x, p.y))
distance_squared_from_origin = p.x * p.x + p.y * p.y
```

- The first line outputs (x=3, y=4). The second line calculates the value 25.

IMPROVING OUR INITIALIZER

- To create a point at position (7, 6) currently needs three lines of code:

```
p = Point()
p.x = 7
p.y = 6
```

- We can make our class constructor more general by placing extra parameters into the __init__ method, as shown in this example:

```
class Point:
    """ Point class represents and manipulates x,y coords. """
    def __init__(self, x=0, y=0):
```

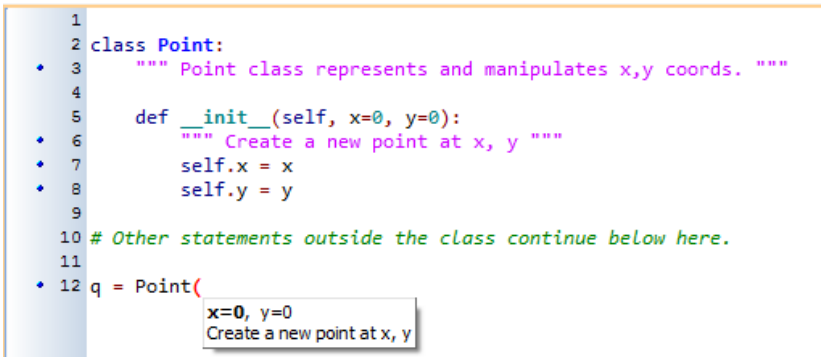
```
        """ Create a new point at x, y """
        self.x = x
        self.y = y
    # Other statements outside the class continue below here.
```

- The x and y parameters here are both optional. If the caller does not supply arguments, they'll get the default values of 0. Here is our improved class in action:

```
>>> p = Point(4, 2)
>>> q = Point(6, 3)
>>> r = Point()                    # r represents the origin (0, 0)
>>> print(p.x, q.y, r.x)
4 3 0
```

Technically speaking . . .

If we are really fussy, we would argue that the __init__ method's docstring is inaccurate. __init__ doesn't create the object (i.e. set aside memory for it), — it just initializes the object to its factory-default settings after its creation.
But tools like PyScripter understand that instantiation — creation and initialization — happen together, and they choose to display the initializer's docstring as the tooltip to guide the programmer that calls the class constructor.
So, we're writing the docstring so that it makes the most sense when it pops up to help the programmer who is using our Point class:



ADDING OTHER METHODS TO OUR CLASS

- The key advantage of using a class like Point rather than a simple tuple (6, 7) now becomes apparent. We can add methods to the Point class that are sensible operations for points, but which may not be appropriate for other tuples like (25, 12) which might represent, say, a day and a month, e.g. Christmas day.
- So being able to calculate the distance from the origin is sensible for points, but not for (day, month) data. For (day, month) data, we'd like different operations, perhaps to find what day of the week it will fall on in 2020.
- Creating a class like Point brings an exceptional amount of "organizational power" to our programs, and to our thinking. We can group together the sensible operations, and the kinds of data they apply to, and each instance of the class can have its own state.

- A method behaves like a function but it is invoked on a specific instance, e.g. tess.right(90). Like a data attribute, methods are accessed using dot notation. Let's add another method, distance_from_origin, to see better how methods work:

```
class Point:
    """ Create a new Point, at coordinates x, y """

    def __init__(self, x=0, y=0):
        """ Create a new point at x, y """
        self.x = x
        self.y = y
    def distance_from_origin(self):
        """ Compute my distance from the origin """
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5
```

- Let's create a few point instances, look at their attributes, and call our new method on them: (We must run our program first, to make our Point class available to the interpreter.)

```
>>> p = Point(3, 4)
>>> p.x
3
>>> p.y
4
>>> p.distance_from_origin()
5.0
>>> q = Point(5, 12)
>>> q.x
5
>>> q.y
12
>>> q.distance_from_origin()
13.0
>>> r = Point()
>>> r.x
0
>>> r.y
0
>>> r.distance_from_origin()
0.0
```

- When defining a method, the first parameter refers to the instance being manipulated. As already noted, it is customary to name this parameter self.
- Notice that the caller of distance_from_origin does not explicitly supply an argument to match the self parameter— this is done for us, behind our back.

Review questions:

1. What is an attribute in a Python object?
2. What is the purpose of dot notation in accessing attributes?
3. What is the role of the self-parameter in a class method?

4. Why does the caller not need to provide an argument for `self` when calling an instance method?

Lecture - 32

INSTANCES AS ARGUMENTS AND PARAMETERS

- We can pass an object as an argument in the usual way. We've already seen this in some of the turtle examples, where we passed the turtle to some function like draw_bar in the chapter titled Conditionals, so that the function could control and use whatever turtle instance we passed to it.
- Be aware that our variable only holds a reference to an object, so passing tess into a function creates an alias: both the caller and the called function now have a reference, but there is only one turtle!
- Here is a simple function involving our new Point objects:

```
def print_point(pt):
    print("({0}, {1})".format(pt.x, pt.y))
```

- print_point takes a point as an argument and formats the output in whichever way we choose. If we call print_point(p) with point p as defined previously, the output is (3, 4).

CONVERTING AN INSTANCE TO A STRING

- Most object-oriented programmers probably would not do what we've just done in print_point. When we're working with classes and objects, a preferred alternative is to add a new method to the class. And we don't like chatterbox methods that call print.
- A better approach is to have a method so that every instance can produce a string representation of itself. Let's initially call it to_string:

```
class Point:
    # ...
    def to_string(self):
        return "({0}, {1})".format(self.x, self.y)
```

- Now we can say:

```
>>> p = Point(3, 4)
>>> print(p.to_string())
(3, 4)
```

- But don't we already have a str type converter that can turn our object into a string? Yes! And doesn't print automatically use this when printing things? Yes again! But these automatic mechanisms do not yet do exactly what we want:

```
>>> str(p)
'<__main__.Point object at 0x01F9AA10>'
>>> print(p)
'<__main__.Point object at 0x01F9AA10>'
```

- Python has a clever trick up its sleeve to fix this. If we call our new method __str__ instead of to_string, the Python interpreter will use our code whenever it needs to convert a Point to a string. Let's re-do this again, now:

```
class Point:
```

```
    # ...

    def __str__(self): # All we have done is renamed the method
        return "({0}, {1})".format(self.x, self.y)
```

- Now things are looking great!

```
>>> str(p)          # Python now uses the __str__ method that we wrote.
(3, 4)
>>> print(p)
(3, 4)
```

INSTANCES AS RETURN VALUES

- Functions and methods can return instances. For example, given two Point objects, find their midpoint. First we'll write this as a regular function:

```
def midpoint(p1, p2):
    """ Return the midpoint of points p1 and p2 """
    mx = (p1.x + p2.x)/2
    my = (p1.y + p2.y)/2
    return Point(mx, my)
```

- The function creates and returns a new Point object:

```
>>> p = Point(3, 4)
>>> q = Point(5, 12)
>>> r = midpoint(p, q)
>>> r
(4.0, 8.0)
```

- Now let us do this as a method instead. Suppose we have a point object, and wish to find the midpoint halfway between it and some other target point:

```
class Point:
    # ...
    def halfway(self, target):
        """ Return the halfway point between myself and the target """
        mx = (self.x + target.x)/2
        my = (self.y + target.y)/2
        return Point(mx, my)
```

- This method is identical to the function, aside from some renaming. It's usage might be like this:

```
>>> p = Point(3, 4)
>>> q = Point(5, 12)
>>> r = p.halfway(q)
>>> r
(4.0, 8.0)
```

- While this example assigns each point to a variable, this need not be done. Just as function calls are composable, method calls and object instantiation are also composable, leading to this alternative that uses no variables:

```
>>> print(Point(3, 4).halfway(Point(5, 12)))
(4.0, 8.0)
```

Review questions:

1. What happens when you pass an object to a function—does it create a new object?
2. What is the purpose of the to_string method in a class?
3. What is the halfway method used for in the Point class?
4. What does the __str__ method do for a class instance?

Question Bank

1. Explain the concept Random numbers with an example.
2. Illustrate time module with an example.
3. Outline the math module with an example.
4. How are namespaces, files and modules related?
5. Differentiate Mutable versus Immutable and Aliasing.
6. Explain user-defined compound datatypes.
7. How to add other methods to the class?
8. Demonstrate the instances as arguments and parameters with an example.
9. How to convert an instance to a string?