Lecture 2 - Python Part 2

Instructor: Suresh Melvin Sigera
Email: suresh.sigera@cuny.csi.edu

Agenda

- Python Control Flow
  - Conditions
  - Iteration
- Classes, Functions, and Modules
  - Functions, Scope
  - Arguments, Pass by Assignment
  - Classes, Methods
- QA

Conditions

The **if** statement has the following general form.

```
if expression:
    statements
```

If the boolean expression evaluates to **True**, the statements are executed. Otherwise, they are skipped entirely.

Conditions

```
a = 1
b = 0
if a:
   print ("a is true!")
if not b:
   print ("b is false!")
if a and b:
   print ("a and b are true!")
if a or b:
   print ("a or b is true!")
```

Conditions

```
a = 1
b = 0
if a:
   print ("a is true!")
if not b:
   print ("b is false!")
if a and b:
   print ("a and b are true!")
if a or b:
   print ("a or b is true!")
```

```
a is true!
b is false!
a or b is true!
```

Conditions

Conditional programming, or branching, is something you do every day, every moment. It's about evaluating conditions: if the light is green, then I can cross; if it's raining, then I'm taking the umbrella; and if I'm late for work, then I'll call my manager.

Conditions

```
late = True
if late:
    print('I need to call my manager!')
```

This is possibly the simplest example: when fed to the if statement, **late** acts as a conditional expression, which is evaluated in a Boolean context (exactly like if we were calling bool(late)). If the result of the evaluation is True, then we enter the body of the code immediately after the if statement. Notice that the **print instruction is indented: this means it belongs to a scope defined by the if clause**. Execution of this code yields:

Conditions

This time I set late = False, so when I execute the code, the result is different.

```
late = False
if late:
   print('I need to call my manager!')
else:
   print('no need to call my manager...')
```

A specialized else – elif

Sometimes all you need is to do something if a condition is met (a simple if clause). At other times, you need to provide an alternative, in case the condition is False (if/else clause), but there are situations where you may have more than two paths to choose from, so, since calling the manager (or not calling them) is kind of a binary type of example (either you call or you don't), let's change the type of example and keep expanding. This time, we decide on tax percentages. If my income is less than $10,000, I won't pay any taxes. If it is between $10,000 and $30,000, I'll pay 20% in taxes. If it is between $30,000 and $100,000, I'll pay 35% in taxes, and if it's over $100,000, I'll (gladly) pay 45% in taxes.

Let's put this all down into beautiful Python code.

A specialized else – elif

```
income = 15000
if income < 10000:
   tax_coefficient = 0.0  #1
elif income < 30000:
   tax_coefficient = 0.2  #2
elif income < 100000:
   tax_coefficient = 0.35  #3
else:
   tax_coefficient = 0.45  #4
print('I will pay:', income * tax_coefficient, 'in taxes')
```

Iteration

If you have any experience with looping in other programming languages, you will find Python's way of looping a bit different. First of all, what is looping? Looping means being able to repeat the execution of a code block more than once, according to the loop parameters we're given. There are different looping constructs, which serve different purposes, and Python has distilled all of them down to just two, which you can use to achieve everything you need. These are the **for** and **while** statements.

Iteration – for loop

The for loop is used when looping over a sequence, such as a list, tuple, or a collection of objects. Let's start with a simple example and expand on the concept to see what the Python syntax allows us to do.

```
for number in [0, 1, 2, 3, 4]:
    print(number)
```

Iteration – for loop

```
for number in [0, 1, 2, 3, 4]:
    print(number)
```

This simple snippet of code, when executed, prints all numbers from 0 to 4. The for loop is fed the list **[0, 1, 2, 3, 4]** and at each iteration, number is given a value from the sequence (which is iterated sequentially, in order), then the body of the loop is executed (the print line). The number value changes at every iteration, according to which value is coming next from the sequence. When the sequence is **exhausted**, the for loop terminates, and the execution of the code resumes normally with the code after the loop.

Iterating over a range

Sometimes we need to iterate over a range of numbers, and it would be quite unpleasant to have to do so by **hard-coding** the list somewhere. In such cases, the range function comes to the rescue. Let's see the equivalent of the previous snippet of code.

```
for number in range(5):
    print(number)
```

Iterating over a range

The **range function** is used extensively in Python programs when it comes to creating sequences: you can call it by passing one value, which acts as stop (counting from 0), or you can pass two values (start and stop), or even three (start, stop, and step). For the moment, ignore that we need to wrap range(...) within a list. The range object is a little bit special, but in this case, we're just interested in understanding what values it will return to us. You can see that the deal is the same with slicing: start is included, stop excluded, and optionally you can add a step parameter, which by default is 1.

Check out the example in the next slide.

Iterating over a range

```
>>>list(range(10)) # 1 value: from 0 to value (excluded)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>list(range(3, 8)) #  2 values: from start to stop (excluded) [3, 4, 5, 6, 7]
>>>list(range(-10, 10, 4)) # 3 values: step is added [-10, -6, -2, 2, 6]
```

Iterating over a sequence

Now we have all the tools to iterate over a sequence, so let's build on that example.

```
surnames = ['Rivest', 'Shamir', 'Adleman']
        for position in range(len(surnames):
                print(position, surnames)
```

Iterating over a sequence

The preceding code adds a little bit of complexity to the game. Execution will show this result.

0 Rivest
1 Shamir
2 Adleman

Iterating over a sequence

Let's use the inside-out technique to break it down. We start from the innermost part of what we're trying to understand, and we expand outward. So, len(surnames) is the length of the surnames list: 3. Therefore, range(len(surnames)) is actually transformed into range(3). This gives us the range [0, 3), which is basically a sequence (0, 1, 2). This means that the for loop will run three iterations. In the first one, position will take value 0, while in the second one, it will take value 1, and finally value 2 in the third and last iteration. What is (0, 1, 2), if not the possible indexing positions for the surnames list?

At position 0, we find 'Rivest', at position 1, 'Shamir', and at position 2, 'Adleman'. If you are curious about what these three men created together, change print(position, surnames[position]) to print(surnames[position][0], end=''), add a final print() outside of the loop, and run the code again.

Iterating over a sequence

Now, this style of looping is actually much closer to languages such as Java or C++. In Python, it's quite rare to see code like this. You can just iterate over any sequence or collection, so there is no need to get the list of positions and retrieve elements out of a sequence at each iteration. It's expensive, needlessly expensive. Let's change the example into a more Pythonic form.

```
surnames = ['Ainsley', 'Harley', 'Marston', 'Melvin']
        for surname in surnames:
                print(surnames)
```

Iterating over a sequence

Now that's something! It's practically English. The for loop can iterate over the surnames list, and it gives back each element in order at each interaction. Running this code will print the four surnames, one at a time, and it is much easier to read.

Iterating over a sequence

What if you wanted to print the position as well though? Or what if you actually needed it? Should you go back to the range(len(...)) form? **No**. You can use the **enumerate** built-in function, like this.

```
surnames = ['Ainsley', 'Harley', 'Marston', 'Melvin']
    for position, surname in enumerate(surnames):
        print(position, surnames)
```

Iterating over a sequence

This code is very interesting as well. Notice that enumerate gives back a two-tuple (position, surname) at each iteration, but still, it's much more readable (and more efficient) than the **range**(**len**(...)) example. You can call enumerate with a start parameter, such as **enumerate**(iterable, start), and it will start from start, rather than 0. Just another little thing that shows you how much thought has been given in designing Python so that it makes your life easier.

You can use a for loop to iterate over lists, tuples, and in general anything that Python calls iterable.

Iterating over multiple sequences

Let's see another example of how to iterate over two sequences of the same length, in order to work on their respective elements in pairs. Say we have a list of people and a list of numbers representing the age of the people in the first list. We want to print a pair person/age on one line for all of them.

Let's start with an example and let's refine it gradually.

Iterating over a sequence

```
people = ['Conrad', 'Deepak', 'Heinrich', 'Tom’]
ages = [29, 30, 34, 36]
for position in range(len(people)):
        person = people[position]
        age = ages[position]
        print(person, age)
```

Iterating over a sequence

By now, this code should be pretty straightforward for you to understand. We need to iterate over the list of positions (0, 1, 2, 3) because we want to retrieve elements from two different lists. Executing it we get the following.

Conrad 29
Deepak 30
Heinrich 34
Tom 36

Iterating over a sequence

This code is both **inefficient** and not **Pythonic**. It's inefficient **because retrieving an element given the position can be an expensive operation**, and **we're doing it from scratch at each iteration**. The postal worker doesn't go back to the beginning of the road each time they deliver a letter, right? They move from house to house. From one to the next one. Let's try to make it better using **enumerate**.

Iterating over multiple sequences

```
people = ['Conrad', 'Deepak', 'Heinrich', 'Tom’]
ages = [29, 30, 34, 36]
for position, person in enumerate(people):
        age = ages[position]
        print(person, age)
```

Iterating over multiple sequences

That's better, but still not perfect. And it's still a bit ugly. We're iterating properly on people, but we're still **fetching age using positional indexing**, which we want to lose as well.

Iterating over multiple sequences

```
people = ['Conrad', 'Deepak', 'Heinrich', 'Tom’]
ages = [29, 30, 34, 36]
for p, a in zip(people, ages):
        print(p, a)
```

Iterating over multiple sequences

Ah! So much better! Once again, compare the preceding code with the first example and admire Python's elegance. The reason I wanted to show this example is twofold. On the one hand, I wanted to give you an idea of how shorter code in Python can be compared to other languages where the syntax doesn't allow you to iterate over sequences or collections as easily. And on the other hand, and much more importantly, notice that when the for loop asks **zip**(sequenceA, sequenceB) for the next element, it gets back a tuple, not just a single object. It gets back a tuple with as many elements as the number of sequences we feed to the zip function.

Let's expand a little on the previous example in two ways, using explicit and implicit assignment.

Iterating over multiple sequences

```
people = ['Conrad', 'Deepak', 'Heinrich', 'Tom’]
ages = [29, 30, 34, 36]
nationalities = ['Poland', 'India', 'South Africa', 'England’]
for p, a,n in zip(people, ages,nationalities):
        print(p, a,n)
```

Iterating over multiple sequences

In the preceding code, we added the nationalities list. Now that we feed three sequences to the **zip** function, the for loop gets back a **three-tuple** at each iteration. **Notice that the position of the elements in the tuple respects the position of the sequences in the zip call**. The result is.

Conrad 29 Poland
Deepak 30 India
Heinrich 34 South Africa
Tom 36 England

Iterating over multiple sequences

Sometimes, for reasons that may not be clear in a simple example such as the preceding one, you may want to explode the **tuple** within the body of the **for** loop. If that is your desire, it's perfectly possible to do so.

Iterating over multiple sequences

```
people = ['Conrad', 'Deepak', 'Heinrich', 'Tom’]
ages = [29, 30, 34, 36]
nationalities = ['Poland', 'India', 'South Africa', 'England']
for data in zip(people, ages, nationalities):
        p, a, n = data
        print(p, a, n)
```

Iterating over multiple sequences

Here, the three-tuple data that comes from **zip**(...) is exploded within the body of the for loop into three variables: p, a, and n.

The while loop

The **while** loop is similar to the **for** loop, in that they both loop, and at each iteration they execute a body of instructions. What is different between them is that the while loop doesn't loop over a sequence (it can, but you have to write the logic manually and it wouldn't make any sense, you would just want to use a for loop), rather, it loops as **long as a certain condition is satisfied**. When the condition is no longer satisfied, the loop ends.

The while loop

```
people = ['Conrad', 'Deepak', 'Heinrich', 'Tom']
ages = [29, 30, 34, 36]
position = 0
while position < len(people):
    person = people[position]
    age = ages[position]
    print(person, age)
    position += 1
```

Functions

A function is a sequence of instructions that perform a task, **bundled** as a unit.

- They reduce code duplication in a program. By having a specific task taken care of by a nice block of packaged code that we can import and call whenever we want, we don't need to duplicate its implementation
- They help in splitting a complex task or procedure into smaller blocks, each of which becomes a function
- They hide the implementation details from their users
- They improve traceability
- They improve readability

Function definition

```
def name(parameter1, parameter2, . . .):
    body
```

Python function was introduced by **def**, a name, and a parameter list, followed by an indented block of code. Typically a function returns a value.

Calling a function

```
def addMe2Me(x):
    print('apply + operation to the argument')
    return (x + x)
```

This function, presumably meaning “addMe2Me” takes an **object**, adds its current value to itself and returns the sum. While the results are fairly obvious with numerical arguments, we point out that the plus sign works for almost all types. In other words, most of the standard types support the + operator, whether it be numeric addition or sequence concatenation.

Calling a function

```
>>>addMe2Me(4.25)
8.5
>>>addMe2Me(10)
20
>>>addMe2Me('Python')
'PythonPython'
```

Calling a function

```
>>>addMe2Me([-1, 'abc’])
[-1, 'abc', -1, 'abc’]
>>>x = addMe2Me([1,2,1,1])
>>>x
>>>[1, 2, 1, 1, 1, 2, 1, 1]
```

Default arguments

Functions may have parameters which have default values. If present, arguments will take on the appearance of assignment in the function declaration, but in actuality, it is just the syntax for default arguments and indicates that if a value is not provided for the parameter, it will take on the assigned value as a default.

Default arguments

```
def foo(debug=1):
    'determine if in debug mode with default argument'
    if debug:
        print('in debug mode')
    print('done')
```

Default arguments

```
>>>foo()
in debug
mode done
>>>foo(0)
done
```

Default arguments

In the example above, the debug parameter has a default value of 1. When we do not pass in an argument to the function foo(), debug automatically takes on a true value of 1.

On our second call to foo(), we deliberately send an argument of 0, so that the default argument is not used.

*args

The syntax is to use the symbol * to take in a variable number of arguments; by convention, it is often used with the word args. What *args allows you to do is take in more arguments than the normal of formal arguments that you previously defined. With *args, any number of extra arguments can be tacked on to your current formal parameters (including zero extra arguments).

Here's an example which should make this clear:

*args

```
def loadcats(owner, *allcats):
    print('Owner name: ', owner)
    for cat in allcats:
        print(cat)

cats = ['Oscar', 'Max', 'Tiger', 'Sam', 'Misty', 'Simba', 'Coco', 'Chloe', 'Lucy']
loadcats('James', cats)
```

*args

Owner name: James
['Oscar', 'Max', 'Tiger', 'Sam', 'Misty', 'Simba', 'Coco', 'Chloe', 'Lucy']

The * parameter name is arbitrary - you can make whatever you like. For an example, I have used *allcats but it is customary (but not required) to name it *args.

*args

```
def loadcats(owner, *args):
    print('Owner name: ', owner)
    for cat in args:
        print(cat)

cats = ['Oscar', 'Max', 'Tiger', 'Sam', 'Misty', 'Simba', 'Coco', 'Chloe', 'Lucy']
loadcats('James', cats)
```

*args

Once you pass *args it **exhausts the positional arguments. You cannot add more positional arguments** after *args.

```
cats = ['Oscar', 'Max', 'Tiger', 'Sam', 'Misty', 'Simba', 'Coco', 'Chloe', 'Lucy']
loadcats('James', cats, 'Holly')
```

**kwargs

**kwargs is used to pass variable ‘key-value’ arguments to a function, In other words, it passes a dictionary as an argument. As it is a dictionary, all dictionary operations can be performed. We can pass any number of key-value pairs.

Here’s an example which should make this clear:

**kwargs

```
students = {
   1:
     {
        'name': 'John',
        'gpa': 3.0,
        'classes_taken': ('CSC126', 'CSC211', 'CSC326')
     },
   2:
     {
        'name': 'Mike',
        'gpa': 2.0,
        'classes_taken': ('CSC126', 'CSC211', 'CSC326')
     },
}
```

**kwargs

```
def load_student(kwargs):
    for key, values in kwargs.items():
        print('\n')
        for value in values:
            print(values[value])
```

**kwargs

John
3.0
('CSC126', 'CSC211', 'CSC326')

Mike
2.0
('CSC126', 'CSC211', 'CSC326')

*args and **kwargs

```
def func(a, b, c=7, *args, **kwargs):
    print('a, b, c:', a, b, c)
    print('args:', args)
    print('kwargs:', kwargs)


func(1, 2, 3, *(5, 7, 9), **{'A': 'a', 'B': 'b'})
func(1, 2, 3, 5, 7, 9, A='a', B='b')  # same as the line above
```

*args and **kwargs

From our previous code, the output as follows:

```
a, b, c: 1 2 3
args: (5, 7, 9)
kwargs: {'A': 'a', 'B': 'b'}
a, b, c: 1 2 3
args: (5, 7, 9)
kwargs: {'A': 'a', 'B': 'b'}
```

Returning a value from a function

Python recognizes the end of a function by the indenting—or more accurately, by the lack of indenting. As soon as a line is found that has the same indenting as the def statement that started the definition of the function, Python knows that it has reached the end of the definition of a function. After the last indented statement runs, control passes back to a point just past where the function was called.

In Python, when a function wants to give a result to a caller, it uses a return statement and specifies the value to hand back. The generic form looks like this.

return <list of values>

Returning a value from a function

The caller can use the resulting value for whatever it needs. Often, the caller will take the resulting value and store it in a variable. For example, here is a modified version of the previous addTwo function that returns a single number value.

Returning a value from a function

```
def addTwo(startingValue):
    endingValue = startingValue + 2
    return endingValue # returns a result to the caller

sum1 = addTwo(5)
sum2 = addTwo(10)

print('The results of adding 2 to 5 and 2 to 10 are:', sum1, 'and', sum2)
```

Returning a value from a function

In this example, we first call the addTwo function with an argument of 5. Inside the function, that value is assigned to the startingValue parameter variable. The function runs and calculates an endingValue of 7. Using a return statement, the function hands back a result to the caller. In the assignment statement in the main code, the value of the call addTwo(5) becomes 7, the rest of the assignment statement runs, and the variable sum1 is set to 7. The second call then runs the same way, and the variable sum2 is set to 12.

Returning a value from a function

Python has a further extension of the return statement. In most other programming languages, the return statement can only return either no values or a single value.

```
def myFunction(parameter1, parameter2):
    # Body of the function, calculates
    # values for answer1, answer2, and answer3
    # hand back three answers to the caller
    return answer1, answer2, answer3
```

Returning a value from a function

Then you would call myFunction with code like this.

```
variable1, variable2, variable3 = myFunction(argument1, argument2)
```

The main function

Python main function is executed only when it’s being executed as a python program. If you don’t have a main function in your program, upon execution Python will provide the main function for you.

The main function

```
def sample():
   print ('Hello World')

>>sample
<function __main__.sample()>
```

The main function

Before the Python interpreter executes your program, it defines a few special variables. One of those variables is called **__name__** and it is automatically set to the string value **"__main__"** when the program is being executed by itself in a standalone fashion.

On the other hand, if the program is being imported by another program, then the **__name__** variable is set to the name of that module. This means that we can know whether the program is being run by itself or whether it is being used by another program and based on that observation, we may or may not choose to execute some of the code that we have written.

The main function

For example, assume that we have written a collection of functions to do some simple math. We can include a main function to invoke these math functions. It is much more likely, however, that these functions will be imported by another program for some other purpose. In that case, we would not want to execute our main function. The active-code below defines two simple functions and a main.

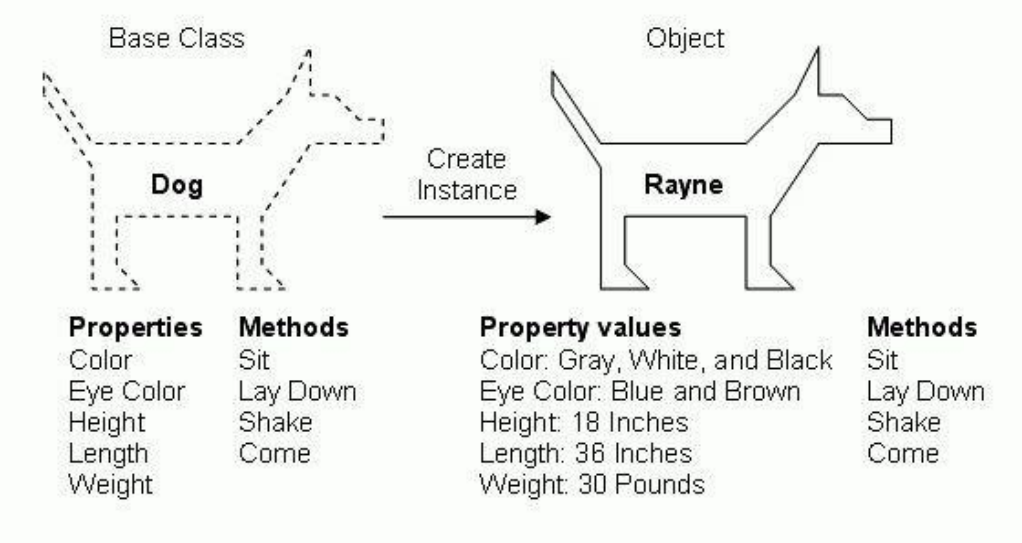
The main function

```
def foo1():
        print('calling foo 1')

def foo2():
        print('calling foo 2')

def main():
        print('calling main')
        foo1()
        foo2()

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

Classes and Objects

A class is a blueprint or template or set of instructions to build a specific type of object. Every object is built from a class. Each class should be designed and programmed to accomplish one, and only one, thing. (You'll learn more about the Single Responsibility Principle in Object-oriented programming concepts: Writing classes.) Because each class is designed to have only a single responsibility, many classes are used to build an entire application.

Classes and Objects

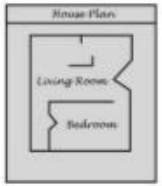


Classes and Objects

Class

3 objects /
instances /
individuals

Blueprint that describes a house



Instances of the house described by the blueprint

Classes and Objects

Think about classes, instances, and instantiation like baking a cake. A class is like a recipe for chocolate cake. The recipe itself is not a cake. You can't eat the recipe (or at least wouldn't want to). If you correctly do what the recipe tells you to do (instantiate it) then you have an edible cake. That edible cake is an instance of the chocolate cake class.

An instance is a specific object built from a specific class. It is assigned to a reference variable that is used to access all of the instance's properties and methods. When you make a new instance the process is called instantiation.

Classes and Objects

You can bake as many cakes as you would like using the same chocolate cake recipe. Likewise, you can instantiate as many instances of a class as you would like. Pretend you are baking three cakes for three friends who all have the same birthday but are different ages. You will need some way to keep track of which cake is for which friend so you can put on the correct number of candles. A simple solution is to write each friend's name on the cake. Reference variables work in a similar fashion. A reference variable provides a unique name for each instance of a class. In order to work with a particular instance, you use the reference variable it is assigned to.

Classes and Objects

Classes can be thought of as **blueprints for creating objects**. When I define a Customer class using the class keyword, I haven't actually created a customer. Instead, what I've created is a sort of instruction manual for constructing "customer" objects. Let's look at the following example code.

https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems/blob/master/week2/class-part-1.3.py

The class Customer(object) line does not create a new customer. That is, just because we've defined a Customer doesn't mean we've created one; we've merely outlined the blueprint to create a Customer object. To do so, we call the class's __init__ method with the proper number of arguments (minus self, which we'll get to in a moment).

Classes and Objects

So, to use the "blueprint" that we created by defining the **class** Customer (which is used to create Customer objects), we call the class name almost as if it were a function:

```
jim = Customer('Jim Carrey', 500000.0)
```

This line simply says "use the Customer blueprint to create me a new object, which I'll refer to as jim." The jim object, known as an instance, is the realized version of the Customer class.

Before we called Customer(), no Customer object existed. We can, of course, create as many Customer objects as we'd like. There is still, however, only one Customer class, regardless of how many instances of the class we create.

Self keyword

So what's with that self parameter to all of the Customer methods? What is it? Why, it's the instance, of course! Put another way, a method like withdraw defines the instructions for withdrawing money from some abstract customer's account. Calling jim.withdraw(100.0) puts those instructions to use on the **jim instance**.

So when we say def withdraw(self, amount):, we're saying, "here's how you withdraw money from a Customer object (which we'll call self) and a dollar figure (which we'll call amount). self is the instance of the Customer that withdraw is being called on. That's not me making analogies, either. jim.withdraw(100.0) is just shorthand for Customer.withdraw(jim, 100.0), which is perfectly valid (if not often seen) code.

__init__

self may make sense for other methods, but what about __init__? When we call __init__, we're in the process of creating an object, so how can there already be a self? Python allows us to extend the self pattern to when objects are constructed as well, even though it doesn't exactly fit. Just imagine that

jim = Customer('Jim Carrey', 1000.0) is the same as calling jim = Customer(jim, 'Jim Carrey', 1000.0) the jim that's passed in is also made the result.

This is why when we call **__init__**, we initialize objects by saying things like self.name = name. Remember, since self is the instance, this is equivalent to saying jim.name = name, which is the same as jim.name = 'jim Carrey. Similarly, self.balance = balance is the same as jim.balance = 1000.0. After these two lines, we consider the Customer object "initialized" and ready for use.

__init__ aka constructor

Be careful what you __init__

After **__init__** has finished, the caller can rightly assume that the object is ready to use. That is, after `jim = Customer('jim Carrey', 1000.0)`, we can start making deposit and withdraw calls on jim; jim is a fully-initialized object.

Creating Python classes

We don't have to write much Python code to realize that Python is a very clean language. When we want to do something, we can just do it, without having to set up a bunch of prerequisite code. The ubiquitous hello world in Python, as you've likely seen, is only one line.

```
class MyFirstClass:
    pass
```

Creating Python classes

Since our first class doesn't actually add any data or behaviors, we simply use the pass keyword on the second line to indicate that no further action needs to be taken.

Creating Python classes

Since our first class doesn't actually add any data or behaviors, we simply use the pass keyword on the second line to indicate that no further action needs to be taken.

Creating Python classes

```
>>> a = MyFirstClass()
>>> b = MyFirstClass()
>>> print(a)
<__main__.MyFirstClass object at 0xb7b7faec>
>>> print(b)
<__main__.MyFirstClass object at 0xb7b7fbac>
```

Creating Python classes

This code instantiates two objects from the new class, named a and b. Creating an instance of a class is a simple matter of typing the class name, followed by a pair of parentheses.

It looks much like a normal function call, but Python knows we're calling a class and not a function, so it understands that its job is to create a new object. When printed, the two objects tell us which class they are and what memory address they live at.

Memory addresses aren't used much in Python code, but here, they demonstrate that there are two distinct objects involved.

Adding attributes

Now, we have a basic class, but it's fairly useless. It doesn't contain any data, and it doesn't do anything. What do we have to do to assign an attribute to a given object?

In fact, we don't have to do anything special in the class definition. We can set arbitrary attributes on an instantiated object using dot notation.

Adding attributes

```
class Point:
    pass

p1 = Point()
p2 = Point()
```

Adding attributes

```
p1.x = 5
p1.y = 4

p2.x = 3
p2.y = 6

print(p1.x, p1.y)
print(p2.x, p2.y)
```

Adding attributes

If we run this code, the two print statements at the end tell us the new attribute values on the two objects.

```
5 4
3 6
```

This code creates an empty Point class with no data or behaviors. Then, it creates two instances of that class and assigns each of those instances x and y coordinates to identify a point in two dimensions. All we need to do to assign a value to an attribute on an object is use the <object>.<attribute> = <value> syntax.

Adding attributes

This is sometimes referred to as dot notation. You have likely encountered this same notation before when reading attributes on objects provided by the standard library or a third-party library. The value can be anything: a Python primitive, a built-in data type, or another object. It can even be a function or another class!

Adding attributes

Now, having objects with attributes is great, but object-oriented programming is really about the interaction between objects. We're interested in invoking actions that cause things to happen to those attributes. We have data; now it's time to add behaviors to our classes.

Let's model a couple of actions on our Point class. We can start with a method called reset, which moves the point to the origin (the origin is the place where x and y are both zero). This is a good introductory action because it doesn't require any parameters.

Methods

In Python, a method is formatted identically to a function. It starts with the def keyword , followed by a space, and the name of the method. This is followed by a set of parentheses containing the parameter list (we'll discuss that self parameter in just a moment), and terminated with a colon.

The next line is indented to contain the statements inside the method. These statements can be arbitrary Python code operating on the object itself and any parameters passed in, as the method sees fit.

Methods

```
class Point:
    def reset(self):
        self.x = 0
        self.y = 0
```

Methods

```
p = Point()
p.reset()
print(p.x, p.y)
```

Methods

```
p = Point()
p.reset()
print(p.x, p.y)
```

Modules

In Python module, the highest-level program organization unit, which packages program code and data for reuse. In concrete terms, modules usually correspond to Python program files (or extensions coded in external languages such as C, Java, or C#).

Each file is a module, and modules import other modules to use the names they define. Modules are processed with two statements and one important function:

Modules

import

Lets a client (importer) fetch a module as a whole

from

Allows clients to fetch particular names from a module

Modules

In short, modules provide an easy way to organize components into a system by serving as self-contained packages of variables known as namespaces. All the names defined at the top level of a module file become attributes of the imported module object. Ultimately, Python's modules allow us to link individual files into a larger program system.

More specifically, from an abstract perspective, modules have at least three roles:

Modules

Code reuse

Modules let you save code in files permanently. Unlike code you type at the Python interactive prompt, which goes away when you exit Python, code in module files is persistent—it can be reloaded and rerun as many times as needed. More to the point, modules are a place to define names, known as attributes, which may be referenced by multiple external clients.

Modules

System namespace partitioning

Modules are also the highest-level program organization unit in Python. Fundamentally, they are just packages of names. Modules seal up names into self-contained packages, which helps avoid name clashes—you can never see a name in another file, unless you explicitly import that file. In fact, everything “lives” in a module—code you execute and objects you create are always implicitly enclosed in modules. Because of that, modules are natural tools for grouping system components.

Modules

Implementing shared services or data

From an operational perspective, modules also come in handy for implementing components that are shared across a system and hence require only a single copy. For instance, if you need to provide a global object that's used by more than one function or file, you can code it in a module that can then be imported by many clients.

Modules

Let's look at the following module.

https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems/tree/master/week2/module-part1.1

Let's do some live coding

Fire up your Anaconda Spider!

QA

Be up to date

Make sure to star this repo, and click on watch for the code updates.

https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems