Developer Operations

Python Overview 3:

Functions, modules, classes

Presentation Overview

- Functions
- Importing modules
- Classes & OO

Function basics

```
def max(x,y) :
    if x > y :
        return x
    else :
        return y
```

myfuncs.py

```
>>> import myfuncs
>>> myfuncs.max(3,5)
5
>>> myfuncs.max('hello', 'there')
'there'
```

Python interpreter

Function names are like any variable

- Functions are objects
- The same reference rules hold for them as for other objects

```
>>> x = 10
>>> x
10
>>> def x () :
...     print ('hello')
>>> x
<function x at 0x619f0>
>>> x()
Hello
>>> x = 'blah'
>>> x
'blah'
```

Default parameters

- Parameters can be assigned default values
- They are overridden if a parameter is given for them
- The type of the default doesn’t limit the type of a parameter

```
>>> def foo(x = 3) :
...     print (x)
...
>>> foo()
3
>>> foo(10)
10
>>> foo('hello')
hello
```

Named parameters

- Call by name
- Any positional arguments must come before named ones in a call

```
>>> def foo (a,b,c) :
...         print (a, b, c)
...
>>> foo(c = 10, a = 2, b = 14)
2 14 10
>>> foo(3, c = 2, b = 19)
3 19 2
```

It’s all objects…

- Everything in Python is really an object
  - We’ve seen hints of this already…
    ```
    "hello".upper()
    list3.append('a')
    dict2.keys()
    ```
  - These look like Java method calls.
  - New object classes can easily be defined in addition to these built-in data-types.

Defining a Class

- A *class* is a special data type which defines how to build a certain kind of object.
- *Instances* are objects that are created which follow the definition given inside of the class

Methods in Classes

- Define a *method* in a *class* by including function definitions within the scope of the class block
- There must be a special first argument ***`self`*** in *all* method definitions
- There is usually a special method called ***`__init__`*** in most classes

A simple class: *student*

```
class student:
    def __init__(self, n, a):
        self.full_name = n
        self.age = a
    def get_age(self):
        return self.age
```

Instantiating Objects

- `__init__` serves as a constructor for the class. It usually does some initialisation work
- The arguments passed to the class name are given to its `__init__()` method
- So, the __init__ method for student is passed “Bob” and 21 and the new class instance is bound to b:

```
b = student("Bob", 21)
```

Constructor: __init__

- An `__init__` method can take any number of arguments.
- Like other functions or methods, the arguments can be defined with default values, making them optional to the caller.

self

- The first argument of every method is a reference to the current instance of the class
- By convention, we name this argument ***`self`***
- In `__init__`, *`self`* refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called

self

- Although you must specify `self` explicitly when *defining* the method, you don't include it when *calling* the method.
- Python passes it for you automatically

Defining a method:

(this code inside a class definition.)

```
def set_age(self, num):
   self.age = num
```

Calling a method:

```
>>> x.set_age(23)
```

Accessing attributes & methods of a class: use "."

```
>>> f = student('Bob Smith', 23)

>>> f.full_name # Access attribute
'Bob Smith'

>>> f.get_age() # Access a method
23
```

Subclasses

- A class can *extend* the definition of another class
  - Allows use (or extension) of methods and attributes already defined in the previous one.
  - New class: *subclass*. Original: *parent*, *ancestor* or *superclass*

- To define a subclass, put the name of the superclass in parentheses after the subclass's name on the first line of the definition.

```
    class part_time_student(student):
```

Definition of a class extending student

```
class student:

    def __init__(self, n, a):
        self.full_name = n
        self.age = a

    def get_age(self):
        return self.age


class part_time_student(student):

    def __init__(self, n, a, e):
        student.__init__(self, n, a)   #Call __init__ for student
        self.employer = e

    def get_age(self):     #Redefines get_age method entirely
        print ("Age: " + str(self.age))
```

Importing modules

- Use classes & functions defined in another file
- A Python module is a file with the same name (plus the *.py* extension)
- Like Java *import*
- Three formats of the command:

```
import somefile
from somefile import *
from somefile import className
```

- The difference? What gets imported from the file and what name refers to it after importing

import...

```
import somefile
```

- *Everything* in somefile.py gets imported.
- To refer to something in the file, append the text “somefile.” to the front of its name:

```
somefile.className.method("abc")
somefile.myFunction(34)
```

Directories for module files

- *Where does Python look for module files?*
- The list of directories where Python will look for the files to be imported is sys.path
- This is just a variable named ‘path’ stored inside the ‘sys’ module
- To add a directory of your own to this list, append it to this list

```
sys.path.append('/my/new/path')
```

Python program layout – "boilerplate"

```
#!/usr/bin/python3

import sys


def main():
    print ('Hello there', sys.argv[1])

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

Specifies which interpreter to use for this program

Distinguishes whether this file is the start point of a program or is an imported module