CS2204

Program Design and Data Structures for Scientific Computing

Announcements

- Project #1 is posted to Blackboard.
  - We will be creating a class that allows us to represent & manipulate DNA strands
  - We will use a string as our underlying storage container

Acknowledgement: many of the following slides were taken from PPT files found at UMBC.edu

Defining Functions

Function definition begins with “def.”

Function name and its arguments.

```
def get_final_answer(filename):
    """Documentation String"""
  line1
    line2
    return total_counter
```

Colon.

The indentation matters…
First line with less indentation is considered to be outside of the function definition.

The keyword ‘return’ indicates the value to be sent back to the caller.

No header file or declaration of types of function or arguments

Calling a Function

- The syntax for a function call is:

```
>>> def myfun(x, y):
        return x * y
>>> myfun(3, 4)
12
```

- Parameters in Python are *Call by Assignment*
  - Old values for the variables that are parameter names are hidden, and these variables are simply made to *refer to* the new values
  - All assignment in Python, including binding function parameters, uses *reference semantics.*

Functions without returns

- *All* functions in Python have a return value, even if no *return* statement inside the code

- Functions without a *return* will return the special value *None*
  - *None* is a special constant in the language
  - *None* is used like *NULL*, *void*, or *nil* in other languages
  - *None* is also logically equivalent to False
  - The interpreter doesn't print *None*

Function overloading? No.

- There is no function overloading in Python
  - Unlike C++/Java, a Python function is specified by its name alone

    The number, order, names, or types of its arguments *cannot* be used to distinguish between two functions with the same name
  - Two different functions can't have ~~the same name~~, even if they have different arguments

(Note: van Rossum playing with function overloading for the future)

Default Values for Arguments

- You can provide default values for a function's arguments
- These arguments are optional when the function is called

```
>>> def myfun(b, c=3, d="hello"):
            return b + c
>>> myfun(5,3,"hello")
>>> myfun(5,3)
>>> myfun(5)
```

All of the above function calls return 8

Keyword Arguments

- You can call a function with some or all of its arguments out of order as long as you specify their names

- You can also just use keywords for a final subset of the arguments.

```
>>> def myfun(a, b, c):
        return a-b
>>> myfun(2, 1, 43)
  1
>>> myfun(c=43, b=1, a=2)
  1
>>> myfun(2, c=43, b=1)
  1
```

Function & Parameter Types

- Functions and parameters don't have types

```
>>> def double(x):
        return 2 * x

>>> print double(2)
  4
>>> print double(2.2)
  4.4
>>> print double('two')
  twotwo
```

Only use this when the function's behavior depends *only* on properties that all possible arguments share

Functions are first-class objects

Functions can be used as any other datatype, e.g.:

- Arguments to function
- Return values of functions
- Assigned to variables
- Parts of tuples, lists, etc.

```
>>> def square(x):
        return x*x

>>> def applier(q, x):
        return q(x)

>>> applier(square, 7)
  49
```

- Now to explore how to define our own classes & objects in Python…

It's all objects...

- Everything in Python is really an object.
  - We've seen hints of this already…
    ```
    "hello".upper()
    dateStr.split("/")
    ```
  - These look like Java or C++ method calls.
  - New object classes can easily be defined in addition to these built-in data-types.

- In fact, programming in Python is typically done in an object oriented fashion.

Defining a Class

- A *class* is a special data type which defines how to build a certain kind of object.
- The *class* also stores some data items that are shared by all the instances of this class
- *Instances* are objects that are created which follow the definition given inside of the class
- Python doesn't use separate class interface definitions as in some languages
- You just define the class and then use it

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* of method definitions which gets bound to the calling instance
- There is usually a special method called ***__init__*** in most classes
- We'll talk about both later…

A simple class def: *student*

```
class student:
   """A class representing a student"""

   def __init__(self,n,a):
       self._full_name = n
       self._age = a

   def get_age(self):
       return self._age
```

Note Python convention of starting instance variable names with an underscore.

Instantiating Objects

- There is no “new” keyword as in Java.
- Just use the class name with ( ) notation and assign the result to a variable
- `__init__` serves as a constructor for the class. Usually does some initialization 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.
- However, the first argument `self` in the definition of `__init__` is special…

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
- Similar to the keyword *this* in Java or C++
- But Python uses *self* more often than Java uses *this*

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)
```

Deleting instances: No Need to "free"

- When you are done with an object, you don't have to delete or free it explicitly.
- Python has automatic garbage collection.
- Python will automatically detect when all of the references to a piece of memory have gone out of scope. Automatically frees that memory.
- Generally works well, few memory leaks
- There's also no "destructor" method for classes

Definition of student

```
class student:
  """A class representing a student"""
  def __init__(self,n,a):
      self._full_name = n
      self._age = a

  def get_age(self):
      return self._age
```

Traditional Syntax for Access

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

>>> f._full_name # Access attribute
"Bob Smith"

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

- Let’s review the code being made available for Project #1