LECTURE 2: PYTHON BASICS - MODULES AND FUNCTIONS

Program Execution

- When you type “python3 mycode.py”, the interpreter reads the source code from mycode.py, analyzes the code, and executes the code **line by line.**
  - Some statements perform operations, others bind names with objects (e.g. function definition).
  - Python program runs from the beginning of the file to the end of the file.
    - A C++ program starts from the main function.

```
''' Module fib.py '''

print("I am in Module fib.py")
def   even_fib(n):
    total  =  0
    f1,    f2 =  1, 2
    while  f1 <  n:
        if f1 %  2  == 0:
            total  =  total  +  f1
        f1,    f2 =  f2,    f1 +  f2
    return total

a = "Hello World!"
print(a)

if __name__   == "__main__":
    limit=input("Max Fibonacci number: ")
    print(even_fib(int(limit)))
```

Command Line Arguments

- To access command line arguments in python code:
  - import sys
  - The arguments are in the sys.argv list.
  - See lect2/commandline.py

Modules

- A module is a file containing Python definitions and statements, just like any Python program.
- The file name is the module name with the suffix .py appended.
- See lect2/mymodule.py for example
- A module can be executed directly or imported by other modules

```
''' Module fib.py '''

print("I am in Module fib.py")
def  even_fib(n):
    total  =  0
    f1,   f2 =  1, 2
    while  f1 <  n:
        if f1 %  2  == 0:
            total  =  total  +  f1
        f1,   f2 =  f2,   f1 +  f2
    return total

a = "Hello World!"
print(a)

if __name__   == "__main__":
    limit=input("Max Fibonacci number: ")
    print(even_fib(int(limit)))
```

Modules

- Within a module, the module's name (as a string) is available as the value of the global variable __name__. (*two underscores on both sides!)*
- If a module is executed directly however, the value of the global variable __name__ will be "__main__".
  - You would often see the following structure in a module, what does it do?

    If __name__ == "__main__":

        statements

```
''' Module fib.py '''

print("I am in Module fib.py")
def   even_fib(n):
    total  =  0
    f1,    f2 =  1, 2
    while  f1 <  n:
        if f1 %  2  == 0:
            total  =  total  +  f1
        f1,    f2 =  f2,    f1 +  f2
    return total

a = "Hello World!"
print(a)

if __name__   == "__main__":
    limit=input("Max Fibonacci number: ")
    print(even_fib(int(limit)))
```

Modules

- A module can be imported from another python file.
- See lect2/myimport.py for example. Note the syntax for the import statement (no quote around module name).
- Import in python is similar to '#include' in C++ with some differences
  - The functions and global variables defined within the scope of the module. Global variable aString in myModule that is imported by myimport.py is accessed by **myModule.aString**.
  - Modules can contain executable statements aside from definitions. These are executed only the first time the module name is encountered in an import statement as well as if the file is executed as a script.
    - ❖ Code in a module – functions, global code, etc. is ***evaluated upon import***.

```
import fib
print(fib.even_fib(4000000))
```

EXAMPLE

```
''' Module bar.py '''

print  ("Hi from the top level of " + __name__)

def    print_hello():
    print  ("Hello    from " + __name__)

if __name__ == "__main__":
    print("bar's name is " + __name__)
```

```
''' Module foo.py'''
import bar

print  ("Hi from the top level of " + __name__)

if __name__ == "__main__":
    print("foo's name is main")
    bar.print_hello()
```

What is the output when we do 'python3 bar.py', what about 'python3 foo.py'?

Module search paths

- Look at the code segment on the right, what are the names of the modules that we are importing?
- Where are they?

```
import mymodule
import sys
….
```

Module search paths

- Python will look for the module in the following places, in order:
  - Built-in modules.
  - The directories listed in the sys.path variable. The sys.path variable is initialized from these locations:
    - The current directory.
    - Paths specified in the shell environment variable PYTHONPATH (a list of directory names, with the same syntax as the shell variable PATH).
    - The installation-dependent defaults.
- The sys.path variable may be modified by a Python program to point elsewhere at any time.
- If you install a third-party python package in your own directory, you can set PYTHONPATH to include the path to your package so that python knows where to search for your package.
  - For example, to add /usr/bin to PYTHONPATH,
    - in tcsh, 'setenv PYTHONPATH $PYTHONPATH:/usr/bin'
    - In bash, 'export PYTHONPATH = '$PYTHONPATH:/usr/bin'
    - To find out which shell you are using, do 'echo $SHELL' in the terminal.

```
import mymodule
import sys
….
```

Module search paths

- Let us run the python program (foo.py) with the bar module it imports in a different directory.
  - Let foo.py be at *./run* directory, bar.by at *./modules*
  - *cd run*, and python3 *foo.py,* what happens?

- How to fix the problem to make it work?
  - Update PATHONPATH in shell to include the modules directory and then run python3.

Functions

created with the def keyword

statements in the function
must be indented.

```
def function_name(args):
        statements
```

- *def* is followed by the function name with round brackets enclosing the arguments and a colon
- indented statements form a body of the function
- *return* specifies a list of values to be returned

```
# Defining the function
def print_greeting():
    print ("Hello!")
    print ("How are you today?")

# Calling the function
print_greeting()
```

Example

```
def connect(uname, pword, server,    port):
    print("Connecting  to",   server,    ":",   port,  "...")
    #  Connecting code   here   ...
```

```
#example of calls
connect('admin',   'ilovecats',   'shell.cs.fsu.edu',    9160)
connect('jdoe',    'r5f0g87g5@y', 'linprog.cs.fsu.edu',  6370)
```

- Arguments to a Python function are required and positional

Default argument values

- Provide a default value for any number of arguments in a function
- Allows functions to be called with a variable number of arguments.
- **Arguments with default values must appear at the end of the arguments list!**

```
def connect(uname, pword, server = “localhost”,    port=9160):
    print("Connecting  to",   server,    ":",   port,  "...")
    #  Connecting code   here   ...
```

```
#example of calls
connect('admin',   'ilovecats')
connect('admin',   'ilovecats',   'shell.cs.fsu.edu',    9160)
connect('jdoe',    'r5f0g87g5@y', 'linprog.cs.fsu.edu',  6370)
```

Parameter Passing: Passed by Assignment

- Passed by assignment
  - Each argument is assigned to a variable in the function's scope, which points to the same object that was passed in.
    - Make changes to the object inside a function will change the object outside the function as long as the formal parameter is not assigned to a new object – similar to passed by reference. See lect2/passing0.py and lect2/passing1.py
    - If the variable inside the function is assigned a new object, then the variable is no longer associate with outside object in the calling side and changes will not be reflected outside – different from passed by reference and similar to pass by value. See lect2/passing2.py and lect2/passing3.py.

Functions

What is the output of the following code?

```
def hello_func(names):
    for n in names:
        print ("Hello,", n, "!")
    names[0] = 'Susie'
    names[1] = 'Pete'
    names[2] = 'Will'
names = ['Susan', 'Peter', 'William']
hello_func(names)
print ("The names are now", names, ".")
```

Functions

What is the output of the following code?

```
def hello_func(names):
    for n in names:
        print ("Hello,", n, "!")
    names = ['Susie', 'Pete', 'Will']

names = ['Susan', 'Peter', 'William']
hello_func(names)
print ("The names are now", names, ".")
```

Default Mutable Arguments

- Since Python functions in a module are evaluated upon import, there is *only one reference to a mutable default argument*, created upon import.
- Multiple calls to the function, using the default value for the mutable parameter, will *use the same mutable object*.
- If a value is provided as the actual parameter, a new object is created to hold it, and this won't affect the default object.

- See lect2/defaultmutable1.py and lect2/defaultmutable2.py

Keyword Arguments

- Function calls can change the order of parameters if they specify the name of the argument. This method of calling uses keyword argument (not to be confused with programming language keywords like while)
  - If all arguments are keyword arguments, then the arguments can appear in any order.
  - **If positional arguments are used with keyword arguments, they must be used before keyword arguments.**
- For example, consider the function:

```
def connect(uname, pword, server = 'localhost', port = 9160):
```

- You can make the call

```
connect(uname = 'me', server = 'www.cs.fsu.edu', port = '80', pword = 'APassword')
```

Here, the arguments are specified as key-value pair with their names as the keys.

Example

- Given the following function signature, which of the following calls are valid?

```
def connect(uname, pword, server = 'localhost', port = 9160):
    # connecting code
```

1. `connect('admin', 'ilovecats', 'shell.cs.fsu.edu‘)`
2. `connect(uname='admin', pword='ilovecats', 'shell.cs.fsu.edu')`
3. `connect('admin', 'ilovecats', port=6379, server='shell.cs.fsu.edu')`

Variable Number of Arguments

- Python functions can take a variable number of parameters.
  - If you do not how many arguemets are there in a function, you can use **args (a * before a name)* to create a function that can take any number of parameters
  - See lect2/var_args0.py
  - This can handle any number **positional** arguments.

```
def myFunc(*argv):
    print("myFunc: ", argv)s
    for i in range(0, len(argv))
        print ("argv[", i, "] = ', argv[i])

myfunc()
myFunc(1, 2)
myFunc('one', 'two', 'three', 'four')
```

Variable Number of Arguments

- Python functions can take a variable number of parameters.
  - One * before the argument name (e.g. *arg) will match any number of positional variables.
  - We can also have any number of keyword arguments – put two *'s before the argument name
  - See lect2/var_args1.py

```
def myFunc(**argv):
    print("myFunc: ", argv)s
    for key in argv.keys():
        print ("argv[", key, "] = ', argv[key])

myfunc()
myFunc(a=1, b = 2)
myFunc(aaa= 'one', bbb= 'two', ccc='three')
```

Variable Number of Arguments

- Exercise: create a function prototype that takes any number of arguments followed by any number of keyword arguments.

- See var_args2.py
- What about a function prototype that takes any number of keyword argument followed by any number of positional arguments.

Variable Number of Arguments

The most general form:

```
def funcName(required arguments, variable positional arguments, variable keyword arguments)
```

- Example:

```
def example(param1, *args, **kwargs):
        print ("param1: “, param1)
        for arg in args:
                print (arg)
        for key in kwargs.keys():
                print (key, ":", kwargs[key])

example('one' , 'two' , 'three', server='localhost', port=9160)
```

Annotating Functions

- If we wish to “help” programmers use types for function parameters, we can annotate the functions.

```
def func(arg: arg_type, optarg: arg_type = default) -> return_type:
```

- When running the code, you can also inspect the annotations. They are stored in a special .__annotations__ attribute on the function.

Names and Scopes

- Python is a dynamically-typed language.
  - Its scoping is different from a statically typed language like C++
  - It is quite subtle in many cases
- Scopes in Python:
  - Python only has four scopes: local (within a function), extend (in the function inside function situation, the scope of the enclosing function), global (for the whole module), built-in.
    - The smallest scope is local (within a function). What is C++'s smallest scope?

Names and Scopes

- When a name come to existence in Python?
  - Variable – the first time the variable is assigned a value (at run time)
  - Function (and its arguments) and class – after they are defined using *def* and *class*.
  - Module – after it is imported
- The location of the name assignment decides its scope
  - If a value is assigned to a name **anywhere** inside a function, the name has a **local scope**
  - If the assignment is outside any function, then the name has a **global scope**

```
# namesscopes.py
a = 1
def myFunc():
    for i in range(0, 5):
        if (False) :
            k = i
        else:
            k = 100
        print(i, k)
    print(i, k)

b = 20
i=1000
if (True):
    for  kk in range(0, 100):
         ii=kk

print(kk, ii)
myFunc()
```

The LEGB Rule for Python Scope

- LEGB stands for Local, Enclosing, Global, and Built-in
  - Local (or function) scope: the code block/body of any function.
    - ❖ All names inside a function are created at the function call, not at function definition.
    - ❖ Names defined in a function is only visible in the body of the function.
  - Enclosing (or nonlocal) scope: this is a special scope that only exists for nested functions. If the local scope is an inner function, then enclosing scope is the scope enclosing the function.
  - Global (or module) scope: this is the top-most scope in a program or a module.
  - Built-in scope: Names python assigned in the built-in module.
- LEGB Rule: When Python sees a name, it will look the name up in local, enclosing, global, and built-in scope in sequence.
  - If all fails: NameError.
  - See lect2/scope0.py, scope1.py, scope2.py, scope3.py scope4.py

Changing the Default Scoping

- Following the LEGB rule, we cannot assign a value to a global variable inside a function.
  - This is quite common in coding
- The global and nonlocal statement declares a variable inside a function as a global or a nonlocal variable.
  - Modify lect2/scope4.py to make it work.

```
v = 100
def inc():
    global v
    v = v + 1

inc()
print("v = ", v)
```

Functions as the First Class Objects

- Functions are first-class objects in Python.
- This basically means that whatever you can do with a variable, you can do with a function. These include:
  - Assigning a name to it.
  - Passing it as an argument to a function.
  - Returning it as the result of a function.
  - Storing it in data structures.
- More specifically, being a first-class objects means
  - It can be used where an object can be used
  - It can be constructed where an object can be constructed
  - It can be typed like any other objects.
  - Functions in C++ cannot be constructed on the fly!

Function Closures

- As first-class objects, you can wrap functions within functions.
- Outer functions have free variables that are bound to inner functions.
- A closure is a function object that remembers values in enclosing scopes regardless of whether those scopes are still present in memory.
  - Objects are data with associated methods
  - Closures are functions with associated data

```
def make_inc(x):
    def inc(y):
        # x is closed in
        # the definition of inc
        return x + y
    return inc

inc5 = make_inc(5)
inc10 = make_inc(10)

print(inc5(5))  #returns 10
print(inc10(5)) # returns 15
```

Functions Closures

Closures are hard to define so follow these three rules for generating a closure:

1. We must have a nested function (function inside a function).
2. The nested function must refer to a value defined in the enclosing function.
3. The enclosing function must return the nested function.

Decorators

- Wrappers to existing functions.
- You can extend the functionality of existing functions without having to modify them.

```
def say_hello(name):
    return "Hello, " + str(name) + "!"

def p_decorate(func):
    def func_wrapper(name):
        return "<p>" + func(name) + "</p>"
    return func_wrapper

my_say_hello = p_decorate(say_hello)
print(my_say_hello("John"))
# output: <p>Hello, John!</p>
```

Decorators

- So what kinds of things can we use decorators for?
- Timing the execution of an arbitrary function.
- Memoization – cacheing results for specific arguments.
- Logging purposes.
- Debugging.
- Any pre- or post- function processing.

Iterables, Iterators, and Generators

- An iterable is any Python object capable of returning its members one at a time, permitting it to be iterated over in a for-loop.
- An iterable has the following properties:
  - It can be looped over (e.g. lists, strings, files, etc).
  - Can be used as an argument to iter(), which returns an iterator.
  - Must define __iter__() (or __getitem__()).

Iterables, Iterators, and Generators

- Iterator
  - An iterator is an object that contains a countable number of values.
  - An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.
  - An iterator is an object which implements the iterator protocol, which consist of the __iter__() and __next__() methods
- An iterable object can be passed to the iter() method to create an iterator and then use the next() method to get each item of the object.
  - Raise exception StopIteration when reaching the end of the container. See lect2/iterator.py

Iterables, Iterators, and Generators

- The for statement calls the iter() function on the sequence object.
- The iter() call will return an iterator object (as long as the argument has a built-in __iter__ function) which defines next() for accessing the elements one at a time.

```
for item in [1, 2, 3, 4, 5]:
                print(item)
```

Equivalent

```
list1 = [1,2,3,4,5]
it=iter(list1)
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it)) #StopIteration Exce
Raised
```

Generators

- Generator is a way of defining iterator using a simple function notation.
- Generators are functions that use the `yield` statement to return results when they are ready while pausing the function (Python will remember the context of the generator when this happens).
- A generator function can be accessed by using iterator's next() function

```
def gen():
        yield 10
        yield 20
```

```
a = gen()
next(a)
next(a)
next(a)
```

Generators

- Generators are functions that use the `yield` statement to return results and pause the function.
- Even though generators are not technically iterator objects, they can be used wherever iterators are used. It is ideal for generating sequences.

```
def count_generator():
        n = 0
        while True:
                yield n
                n = n + 1
```

```
counter = count_generator()
next(counter)                    # prints 0
myIt=iter(counter)    # sets up an iterator
```

- See lect2/generator.py

Generators and iterators

- Exercise: Write a generator for the Fibonacci number sequence, 1, 2, 3, 5, 8, ….
  (f0=1, f1=2, f2=f0+f1=3, f3=f1+f2=8, and so on)

- Exercise: Write a generator for the sequence i^2 (0, 1, 4, 9, 16, …)

Generators

- A generator function uses the `yield` statement to return results and pause the function.
- A generator function usually has an infinite loop in the function.
- The sequence produced by a generator can be accessed using next()
- Generators utilize *lazy evaluation* – the code to generate the next item is only executed when the next() is called. This is usually more resource efficient than other alternatives (that usually produce the whole sequence first). Thus, generators are the more desirable way for generating a large sequence.