PYTHON – FUNCTIONS

Functions

- The most basic program structure Python provides for maximizing code reuse.
- Functions also provide procedural decomposition.
- And can make a program much easier to understand

Function Related Statements and Expressions

Statement	Examples
Calls	`myfunc('spam', 'eggs', meat=ham)`
def, return	`def adder(a, b=1, *c):` `  return a + b + c[0]`
global	`def changer():` `  global x; x = 'new'`

`def` general format is as follows:

```
def <name>(arg1, arg2,... argN):
   <statements>
   return(<value>)
```

Defining and calling a function:

```
def exp(x,y):
      z=x**y
      return(z)


x=5; y=11
m=exp(x,y)
```

`def` is executable code

- Unlike functions in compiled languages such as C, `def` is an executable statement—your function does not exist until Python reaches and runs the def.
  - How does this effect the location of functions in your programs?
- It is legal (and even occasionally useful) to nest `def` statements inside if/while statements, and even other `def`s.
- In typical operation, `def` statements are coded in module files and are run when module is first imported.

`def` creates an object and assigns it to a name.

- When Python reaches and runs a `def` statement, it generates a new function object and assigns it to the function's name.
- As with all assignments, the function name becomes a reference to the function object.
  - No different from any other type or object!
- The function object can be assigned to other names, stored in a list, and so on.
- Function objects may also have arbitrary user-defined *attributes* attached to them to record data.

```
def foo():
    pass


>>> foo.score = 10
>>> dir(foo)
['__call__', '__class__', '__delattr__', '__dict__',
'__doc__', '__get__', '__getattribute__', '__hash__',
'__init__', '__module__', '__name__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__str__', 'func_closure', 'func_code',
'func_defaults', 'func_dict', 'func_doc',
'func_globals', 'func_name', 'score']


>>> foo.score
10
>>> foo.score += 1
>>> foo.score
11
```

Local Variables

- Variables defined inside a function are by default local.
  - Only visible to code inside the function and exist only while the function runs.
  - This includes arguments that are passed in.
- Local variables disappear when the function exits.
- Only the return statement is saved!

Local Variables

```
def zero(s1,s2):
    print("Start of ZERO s1 is",s1)
    print("Start of ZERO s2 is" ,s2)
    s1=[1,2,3]
    s2=[4,5,6]
    print("\nEnd of ZERO s1 is ", s1)
    print("End of ZERO s2 is ",s2)
    return(s1,s2)


s1="spam"
s2={1:"m","p":"a"}

b=zero(s1,s2)
print("\nreturned values for s1 & s2 ",b)

print("\ns1 is ",s1)
print("s2 is ",s2)
```

Local Variables

```
>>>
Start of ZERO s1 is spam
Start of ZERO s2 is {1: 'm', 'p': 'a'}

End of ZERO s1 is  [1, 2, 3]
End of ZERO s2 is  [4, 5, 6]

returned values for s1 & s2  ([1, 2, 3], [4, 5, 6])

s1 is  spam
s2 is  {1: 'm', 'p': 'a'}
>>>
```

Default values

```
def fun1(x=1,y=2,z=3):
      return(x+y-z)


fun1()
fun1(2)
fun1(2,3,4)
```

`return`

- `return` sends a result object back to the caller.
- When a function is called it is run then returns control to the caller.
- Functions may return a value back to caller with return statement.

`return`

- A function does not have to use the 'return' statement.

- If not; returns 'None‘.

return

```
def concat_strings(a, b):
    str_type = type('')
    if type(a) == str_type and type(b) == str_type:
        return(a + ' ' + b)


print ('strings:' , concat_strings('first',
'second'))
print ('integers:', concat_strings(1,2))
```

```
>>>
strings: first second
integers: None
>>>
```

Dynamic Typing (Polymorphism in Python)

```
>>>
>>> def mult(x,y):
        return(x*y)

>>> mult(5,6)
30
>>> mult('abd',3)
'abdabdabd'
>>>
```

Polymorphism in Python

- Python leaves it up to the objects to do something reasonable for the syntax.
- An operator is just a dispatch mechanism that is interpreted by the objects being processed.
- This is polymorphism: i.e. the meaning of an operation depends on the objects being operated upon.
- Because it's a "dynamically typed language", polymorphism runs rampant in Python.

Polymorphism in Python

- Every operation is a polymorphic operation in Python.

- A single function can be applied to a whole category of object types automatically.
  - As long as those objects support the expected interface (a.k.a. protocol), the function can process them.

Polymorphism in Python

```
def intersect(seq1, seq2):
      res = []
      for x in seq1:
            if x in seq2:
                  res.append(x)
       return(res)
```

ARGS AND KARGS

```
def foo(a,b,c=4,*arg,**karg):
      return a, b, c, arg, karg

foo(1,2)
foo(1,2,3)
foo(1,2,3,4,5,6)
foo(1,2,3,4,5,d=6)
```

Exercise

Write a function (f(a,b)) that applies a list of 3 functions (f1,f2,f3: defined in __main__) to the passed two integers. The list [f1,f2,f3] respectively adds, mults and divides the two ints passed and return the result.

Ex: f(2,3) -> f will in turn iterate through the list, passing (2,3) to each function in the list and then returning the final result.