User-Defined Functions (Units) and Unit Testing

Python 2

home (../handouts.html)

User-Defined Functions are *named code blocks*

The *function block* is executed every time the function's name is *called*.

```
def print_hello():
    print "Hello, World!"

print_hello()             # prints 'Hello, World!'
print_hello()             # prints 'Hello, World!'
print_hello()             # prints 'Hello, World!'
```

When we run this program, we see the greeting printed three times.

One advantage of this is that when we want the greeting printed, we can call the function by name instead of actually printing the statement ourselves. Just as importantly, if we wanted to change our greeting so that it said "Hello, Earth!" instead, we would just change it in the function - we wouldn't have to change it three times.

Function Argument(s)

Any argument(s) passed to a function are *aliased* to variable names inside the function definition.

```
def print_hello(greeting, person):    # 2 strings aliased to objects
                                      # passed in the call

    full_greeting = "{}, {}!".format(greeting, person)
    print full_greeting

print_hello('Hello', 'World')         # pass 2 strings:  prints "Hello, Wo
print_hello('Bonjour', 'Python')      # pass 2 strings:  prints "Bonjour,
print_hello('squawk', 'parrot')       # pass 2 strings:  prints "squawk, p
```

The *return* Statement Returns a Value

Object(s) are returned from a function using the **return** statement.

```
def print_hello(greeting, person):
    full_greeting = greeting + ", " + person + "!"
    return full_greeting

msg = print_hello('Bonjour', 'parrot')   # full_greeting
                                         # aliased to msg

print msg                                # 'Bonjour, parrot!'
```

Dividing a Program into Steps

We should plan our code design around several discrete operations. Each of these will translate to a function.

take input
Input may come from the user's keyboard or program arguments (discussed in a later unit).

validate input (isdigit(), valid choice, etc.)
Since the user may provide "bad" or incorrect input, we must test the input and return an error message if invalid.

normalize input (str->float, etc.)
The user's input may be correct, but not in a form useful to us -- she/he may be asked to input numbers, but those numbers will come to us as strings. In this step we would convert input to a usable form.

read data; normalize data
Data may come from a file, database or other networked resource, and it is usually *not* in the form in which we'd like to use it. In these steps we read the data and then transform / convert data into a usable form. This may involve selecting portions of the data and adding to a container structure (coming soon).

perform calculation(s)
With the data in our preferred form, we can now perform calculations to produce a desired result -- adding numbers togther, examining or transforming strings, etc.

report result
Finally, we need to output the result to screen, a file, a web page, etc. This is the step in which the program's results are displayed to the user or written to a file or database.

Rendering Coding Steps as Functions

Coding steps can be rendered as functions; the "main body" code controls these functions

In this solution to the tip calculator, we have marked each step in the code with an ALL CAPS comment:

```
# TAKE INPUT
bill_amt = raw_input('Please enter the total bill amount:  ')
party_size =  raw_input('Please enter the number in your party:  ')
tip_pct = raw_input('Please enter the desired tip percentage (for example,

# NORMALIZE INPUT
bill_amt = float(bill_amt)
party_size = int(party_size)
tip_pct = float(tip_pct)

# PERFORM CALCULATIONS
tip_amt = bill_amt * tip_pct * .01
total_bill = bill_amt + tip_amt
person_share = total_bill / party_size

#REPORT RESULT
print 'A ' + str(tip_pct) + '% tip ($' + str(tip_amt) + ') was added to th
print 'With ' + str(party_size) + ' in your party, each person must pay $'
```

(There is no read/normalize data step in this example.)

Here is the same solution organized into functions, along with function comments:

```
# CONSTANT values can be used inside functions
TOTAL_QUERY =      'Please enter the total bill amount:  '
PARTY_SIZE_QUERY = 'Please enter the number in your party:  '
TIP_PCT_QUERY =    'Please enter the desired tip percentage (for example,

MSG1 = 'A {}% tip (${}) was added to the bill, for a total of ${}.'
MSG2 = 'With {} in your party, each person must pay ${}.'


def take_input():
    """ take keyboard input """

    bill_amt = raw_input(TOTAL_QUERY)
    party_size =  raw_input(PARTY_SIZE_QUERY)
    tip_pct = raw_input(TIP_PCT_QUERY)

    return bill_amt, party_size, tip_pct


def normalize_input(bill_amt, party_size, tip_pct):
    """ convert user inputs to float and int """

    bill_amt = float(bill_amt)
    party_size = int(party_size)
    tip_pct = float(tip_pct)

    return bill_amt, party_size, tip_pct


def perform_calculations(bill_amt, party_size, tip_pct):
    """ calculate tip amount, total bill and person's share """

    tip_amt = bill_amt * tip_pct * .01
    total_bill = bill_amt + tip_amt
    person_share = total_bill / party_size

    return tip_amt, total_bill, person_share


if __name__ == '__main__':            # 'main body' code
    bill, size, pct = take_input()

    bill_num, size_num, pct_num = normalize_input(bill, size, pct)
```

```
    tip_amt, total_bill, person_share = perform_calculations(bill_num, siz

    print MSG1.format(pct, tip_amt, total_bill)
    print MSG2.format(size, person_share)
```

* The **CONSTANT VALUES** are values that you know will not change during program execution, but might be changed by the programmer at some point in the future. In the above case, we are showing display text. We set these at the top for reference, because they aren't really part of the logic of the code.

* The functions make up the main actions that our code takes. Most of code will be placed within functions.

* The **if __name__ == '__main__':** is a special **if** test that holds the "main body" code. This is necessary so that the testing program can properly call this code's functions without calling the rest of the code.

Special Warning Not to Use "globals" Inside Functions

Functions should only use argument values. They may sometimes use "constant" values defined at the top of the script. They *must not* refer to global variables defined elsewhere in the "main body" code.

arguments are variables passed to functions

local variables are those defined / initialized inside a function

global variables are ones defined / initialized outside of a function

constant variables are those defined in ALL_CAPS at the top of the script. They are not expected to be chnaged or reassigned anywhere in the script.

The name of this program is **doubler.py** (caps in comments are for emphasis)

```
PI = 3.14                # PI is a CONSTANT


def take_input():
    multi = raw_input('please enter a multiplier: ')
    imulti = int(multi)
    return imulti

def multipi(val):          # val is an ARGUMENT
    dblval = PI * val      # dblval is a LOCAL variable
    return dblval


if __name__ == '__main__':
    uval = take_input()
    dval = multipi(uval)   # dval is a GLOBAL variable
    print 'PI * {} = {}'.format(uval, dval)
```

Keep in mind that we could easily have used **imulti** directly inside **doubleit()** -- instead, we *passed* the global to the function. The use of a global inside a function is a cardinal transgression! Why? Because allowing this can introduce errors into code; also, it can render the function *untestable*.

Unit Testing

Unit Tests are programs with functions that test another program's functions.

For our earliest programs in this class, I will supply test code and you will run the tests to ensure that your code is working properly.

Here again is **doubler.py**

```
PI = 3.14                # PI is a constant


def take_input():
    multi = raw_input('please enter a multiplier: ')
    imulti = int(multi)
    return imulti

def multipi(val):        # val is an argument
    dblval = PI * val     # dblval is a local variable
    return dblval


if __name__ == '__main__':
    uval = take_input()
    dval = multipi(uval)   # dval is a global variable
    print 'PI * {} = {}'.format(uval, dval)
```

The tester for **doubler.py** *imports* doubler.py, then tests its function:

```
import unittest                      # unit testing framework
import doubler                       # code we want to test (in doubler.p

class Test(unittest.TestCase):       # special structure called a class

    def test_doublenum(self):
        dpi = doubler.multipi(2)     # call multipi()
        self.assertEqual(dpi, 6.28)  # test result

        qpi = double.multipi(4.0)    # call multipi()
        self.assertEqual(qpi, 12.56) # test result

unittest.main()                      # start the testing
```

* **import** brings in other Python code into our own program. Your test program will import **unittest** and then import the program to be tested.

* The reat of this code, including **class**, **self** and **unittest.TestCase** will be discussed later in the course.

* Eventually we will learn how to conceptualize and write our own tests, and take part in *test-driven development*.