PYTHON FUNCTIONS

Python Crash Course

Part 4

OUTLINE

- Writing our own functions
- Nested functions
- Documentation

FUNCTIONS

So far we've been using functions from the builtin module, which is automatically loaded in most cases.

Occasionally we've accessed functions in other modules using import.

You can learn what methods are available in a module as follows

```
import random

dir(random)
```

It will list all the available methods. Let’s use one of them

```
a = random.random()
```

We'll now look at building our own functions.

BASIC FUNCTION

Functions are a block begun with a function declaration or header:

```
def function_name ():
      # Suite of statements
```

```
# Code calling function_name
```

In a standard script, functions need to be defined before you use them.

EXAMPLE

```
def printit():
      print("hello world")
```

```
printit()   # The function can be called
printit()   #   as many times as you like.
printit()
```

PASSING INFO IN

Functions can be parameterized; that is, made flexible through variables which are set up when the function is called.

Note that the arguments sent in are allocated to the parameter variables.

```
def printit(text):
       print(text)
```

```
printit("hello world")
```

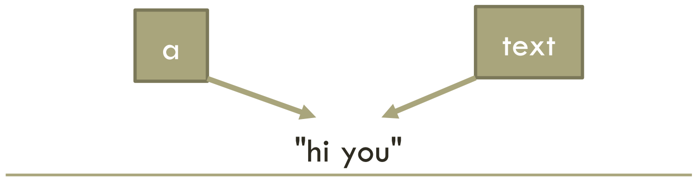
```
a = "hi you"
printit(a)
```

VARIABLE LABELS

What is happening here is we're attaching two labels to the same variable:

```
def printit(text):
      print(text)
```

```
a = "hi you"
printit(a)
```



The effect this has depends whether the variable is mutable or immutable.

VARIABLE LABELS

For mutable variables, changes inside the function change the variable outside:

```
def printit(text):
    text[0] = text[0] + ", Pikachu"
    print(text[0])
```

```
a = ["hi you"]
printit(a)                  # Call the function.
print(a)                    # Check the value of a.
```

As lists are mutable, the value in the list outside the method has changed to "`hi you, Pikachu`".

VARIABLE LABELS

For immutable variables, changes inside the function just create a new variable inside the function. Even though it has the same name, it isn't the same variable:

```
def printit(text):
    text = text + ", Pikachu"   # New text variable created.
    print(text)
```

```
a = "hi you"
printit(a)                      # Call the function.
print(a)                        # Check the value of a.
```

As text is immutable, the value in the list outside the method is still `"hi you"`.

PASSING INFO IN

Python (unlike many languages) doesn't worry about the type passed in

```
def printit(var):
      print(str(var))
```

```
printit("hello world")
printit(430)
printit(430.0)
```

GETTING VALUES BACK

By default, functions invisibly return `None` (you can do so explicitly, as is sometimes useful).

But you can pass values back:

```
def get_pi():
    return 3.14159265359
```

```
pi = get_pi()
print(pi)
```

If something comes back, you either need to attach a label to it, or use it.

You can find out the type of a returned object with `type(variable_name)`.

If you need to return more than one thing, return a tuple.

MULTIPLE IN (SINGLE OUT)

```
def add(num1, num2):
    return num1 + num2
```

```
answer = add(20,30)
```

Arguments are allocated to parameters like this just on position.

These are called positional arguments.

DEFAULTS

You can set up default values if a parameter is missing:

```
def add(num1 = 0, num2 = 0):
    return num1 + num2
```

```
answer = add(3)
```

With this type of parameter, positional arguments are allocated left to right, so here, num1 is 3, and num2 is nothing.

ORDERING

In the absence of a default, an argument *must* be passed in.

If we did this:

```
def add(num1=0, num2):
    return num1 + num2
```

```
answer = add(3)
print(answer)
```

It wouldn't be clear if we wanted num1 or num2 to be 3. Because of this, parameters with defaults must come after any undefaulted parameters. All parameters to the right of a defaulted parameter must have defaults (except * and ** - which we'll see shortly).

KEYWORD ARGUMENTS

You can also name arguments, these are called keyword arguments or kwargs.

Note that this use of "keywords" has nothing to do with the generic use for words you can't use as identifiers.

```
def funct1(num1, num2):
    return 2*num1 + num2
```

```
print(funct1(num2 = 30, num1 = 50))
```

Returns 130

Here we've swapped the order of the positional arguments by naming the parameters to assign their values to.

MIXED ARGUMENTS

```
def add(num1, num2):
    return num1 + num2
```

```
answer = add(3, num2 = 5)
```

Note that once Python has started to deal with kwargs, it won't go back and start allocating positional arguments, so all positional arguments have to be left of the kwargs.

You can force parameters to the right to be kwargs by putting a * in the parameters

Note also that you can't allocate to the same parameter twice:

```
answer = add(3, num1 = 5)
```

*ARGS AND **KWARGS

Mostly used in function definitions.

Allow you to pass a variable number of arguments to a function.

*args is used to send a non-keyworded variable length argument list to the function

**kwargs allows you to pass keyworded variable length of arguments to a function

use **kwargs if you want to handle named arguments in a function

Very useful for APIs

Let's look at them more closely

FLEXIBLE PARAMETERISATION: *ARGS

You can allow for more positional arguments than you have parameters using `*tuple_name`, thus:

```
def sum (num1, num2, *others):
        sum = num1
        sum += num2
        for num in others:
                sum += num
        return sum
answer = sum(1,2,3,4,5,6,7)
```

*is known as the iterable un/packing operator. If nothing is allocated, the tuple is empty.

* is called asterisk (shift 8). Some people also use "star"

ITERABLE UNPACKING

You can equally use the * operator with lists or tuples to generate parameters:

```
def sum (num1, num2, num3, num4):
        return num1 + num2 + num3 + num4
```

```
a = [1,2,3,4]
answer = sum(*a)
```

```
# Note that these can also be in the middle.
a = [10,20]
answer = sum(1,*a, 2)
```

ITERABLE UNPACKING

You can, therefore, use these in both positions:

```
def sum(*nums):
        sum = 0
        for num in nums:
                sum += num
        return sum
```

```
a = [1,2,3]
answer = sum(*a)
```

FLEXIBLE PARAMETERISATION: *KWARGS

The same can be done with `**dict_name` (`**` is the dictionary unpacking operator), which will make a dictionary from unallocated kwargs:

```
def print_details (a, **details):
        first = details["first"]
        surname = details["surname"]
        print (first + " " + surname + " has " + a + " pounds")
```

```
print_details("5", first="George", surname="Formby")
```

Note that you can't use one of these to override other variables. If nothing is allocated, the dictionary is empty.

UNPACKING DICTIONARIES

You can also use `**` to create keyword arguments:

```
def print_details(a, first, surname):
         print (first + " " + surname + " has " + a + " pounds")
```

```
d = {"first":"George","surname":"Formby"}
print_details("5",**d)
```

UNPACKING QUIRKS

Note also that just as with standard arguments `*tuple_name` <u>must come before</u> `**dict_name` if you use both. `*tuple_name` <u>must come after</u> positional parameters and `**dict_name` after kwargs.

It is, therefore usual to place them after their associated variables or together at the end.

```
def func(a,b,*c,d,**e) # d has to be a kwarg.

def func(a,b,d,*c,**e) # abd,bd,or d can be kwargs.
```

But this is also how you force variables to be kwargs.

Note that attempts to do this:

```
def a(**d):

        print(d)

a(d={1:2})
```

End up with a dictionary variable inside the dictionary, with the same name:

```
{d:{1,2}}
```

SUMMARY OF ORDER

Parameters:

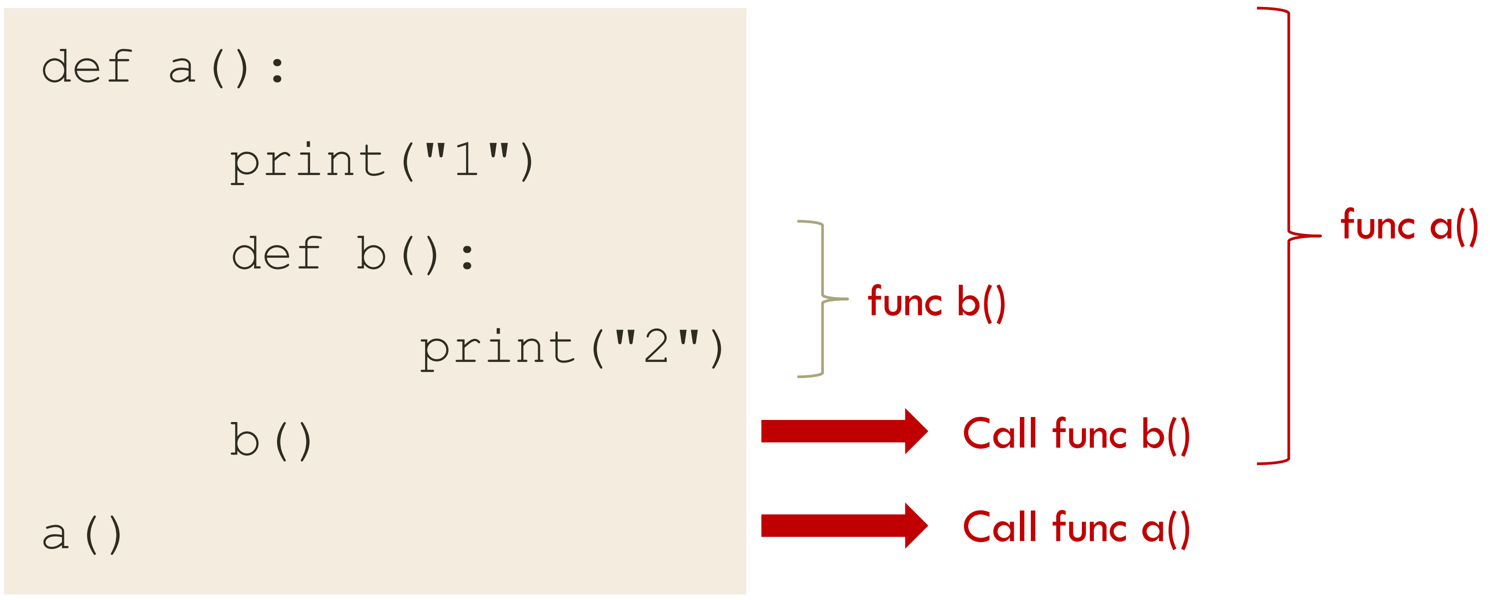
- Undefaulted parameters
- Defaulted parameters (*args)
- *tuple_name
- Forced keyword parameters (**kwargs)
- **dict_name

Arguments:

- Positional arguments
- Keyword arguments
- Unpacked lists, tuples, or dicts may be anywhere

NESTED FUNCTIONS

Note that there is nothing to stop you having functions within functions:



```
def a():
      print("1")
      def b():
            print("2")
      b()
a()
```

SCOPE

Scope is the space within which a variable label exists and can be used. Python talks of names being bound to the code in an area and that determining its scope. Binding in Python is usually through assignment.

So far, we haven't had to deal with it. Loops/if compound statements don't effect it:

```
a = [1]                 # "a" can be declared here
for i in 1,2,3,4:
    a = [1]             # or here.
    a[0] = i
    print (a)           # 1 ,2 ,3 ,4
print (a)               # 4
print (i)
```

All the "a" here are treated as same object wherever they are first used, and "i" can be seen everywhere after it has been first declared.

FUNCTION SCOPE

Functions have more complicated scoping. Although the declaration of a function looks similar to a compound statement clause header, the function declaration generates a block, which has scoping rules.

Starting with variables labels made inside a function:

```
def a ():
        b = 10
        print(b)
a()
print(b)
```

This will run the print statement within the function, but fail for that outside because b has been first allocated inside the function.

This makes b a local variable, only available within the function and functions within it.

FUNCTION SCOPE

One solution in a script is to define the variable outside the function:

```
b = 10

def a ():

      print(b)

a()

print(b)
```

FUNCTION SCOPE

One solution in a script is to define the variable outside the function:

```
b = 10                     # b here is a "global variable"
                           #   it can be see everywhere there
                           #   isn't a variable of the same
                           #      name assigned.
```

```
def a ():
    print(b)               # b here is a "free variable" i.e.
                           #  defined outside the current block.
```

```
a()
print(b)
```

FUNCTION SCOPE

As soon as you declare a variable inside a block, it is a new variable label. Contrast:

```
b = 10
def a ():
         b = 20
         print(b)          # Prints 20.
print(b)                   # Prints 10.
a()
print(b)                   # Prints 10.
```

with:

```
b = 10
def a ():
         print(b)          # Prints 10;
print(b)                   # Prints 10.
a()
print(b)                   # Prints 10.
```

GLOBAL VARIABLES

Variables outside of functions in scripts are global: in theory they can be seen anywhere.

However, the rule about local assignments creating local variables undermines this.

To force a local assignment to a global variable, use the `global` keyword, thus:

```
b = 10
def a ():
       global b
       b = 20
       print(b)       # Prints 20.
print(b)              # Prints 10.
a()
print(b)              # Now prints 20 as the function
                      #   changes the global b.
```

NESTED SCOPES

With nested functions you can imagine situations where you don't want to use a global, but do want a variable across all the functions:

```
a = 1
def f1():
        a = 2
        def f2():
                a = 3
                print(a)        # Prints 3.
        f2()
        print(a)                # Prints 2 - but we'd like 3.
f1()
print(a)                        # Prints 1.
```

NESTED SCOPES

We can achieve this with the `nonlocal` keyword, which propagates variable labels out one nest (unless there they are set to global).

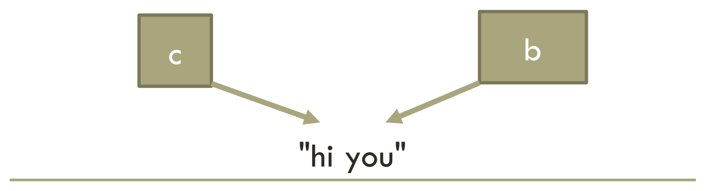
```
a = 1
def f1():
        a = 2
        def f2():
                nonlocal a
                a = 3
                print(a)                # Prints 3.
        f2()
        print (a)                       # Prints 3.
f1()
print(a)                                # Prints 1.
```

VARIABLE LABELS

Note that variable labels in function declarations are also local to the function, but we're only talking about labels here, not values:

```
def a(b):
       print(b)

c = "hi you"
a(c)
```



Here, b is local to the function, but the "hi you" referred to can be seen anywhere there's a label attached to it.

STYLE

Generally style is built into Python, as it demands indents.

However, good Python style is set out in PEP8:

https://www.python.org/dev/peps/pep-0008/

STYLE: MAIN ELEMENTS

Use 4 spaces per indent, rather than tabs.

Use blank lines before function defs, and to separate logical code units.

```
function_names
variable_names
keyword_named_variables_
ClassNames
modname
CONSTANT_NAMES
```

Keep lines to 79 characters or less.

Spaces around operators and after commas, but not just inside `(){}[]`:

```
a = (1, 2)
```

Indent comments to the level of the code referred to.

DOCUMENTATION

Docstrings are automatically extracted to describe your code.

They are triple-double quote enclosed comments after, for example, function declarations.

For some peculiar reason, they are written as commands not descriptions

```
def add(num1, num2):
      """Add two numbers."""
```

EXAMPLE

Style details:
https://www.python.org/dev/peps/pep-0257/

```
def add (num1, num2):
        """
        Add two numbers.


        Keyword arguments:
        num1 -- an integer or double number (no default)
        num2 -- an integer or double number (no default)


        Returns:
        The sum of the numbers.
        """


        return num1 + num2
```

PYDOC

PyDoc is the documentation system distributed with Python.

Best way to invoke it is to import any code, and then type:

```
help(x)
```

Where "x" is a function, module, dotted object method etc.

If you want to see docstrings, then:

```
print(function_name.__doc__)
```

$\uparrow \quad \uparrow$

Two underscores

PYDOC

To generate a webpage from documentation, at command prompt:

```
pydoc -w filename
```

(note without the .py)

For more, see:

https://docs.python.org/3/library/pydoc.html

OTHER SOFTWARE

There is a starter list of alternative software at:

https://wiki.python.org/moin/DocumentationTools

A popular one is Sphinx, which comes with Anaconda:

http://www.sphinx-doc.org/en/stable/

Check tutorials and invocation pages

USING YOUR OWN FUNCTIONS SAVED IN .PY FILES

You can define multiple functions in a single python (.py) file

You can use call these functions both in Jupyter and terminal

Let say the name of the python file is “filename.py”

Name of the function you want to use is “function_name”

```
>>> from filename import function_name
>>> function_name(parameters)
```

QUESTIONS?

EXERCISE TIME