Week 3 - Functions

Functions

Suppose you'd like to print lists of things in the following 'smart' way, which takes into account the number of items and uses commas and 'and' accordingly:

```
A
A and B
A, B, and C
A, B, C, and D
```

You can't just use the `join()` method of a string, because it doesn't do this. You have to write a piece of code. And the code required is reasonably complicated:

```
# Print the list 'lst' in a smart way
if len(lst) == 1:
    print(lst[0])
elif len(lst) == 2:
    print(lst[0] + ' and ' + lst[1])
else:
    print(', '.join(lst[:-1]) + ', and ' + lst[-1])
```

Having to write this code every time you want to smart-print a list would be a pain. Also, if you find an error in the code, or if you think of a way to improve it, you'll have to find all instances of the code and update them one-by-one.

Fortunately, there is a much better way, and that's to define a **function** which does this smart printing. Then whenever you want to smart-print a list you can just call the function and provide it with the list.

Here's how your program might look (don't worry if you don't understand the "def" part - you'll be learning this):

```
# Define the function
def smart_print(lst):
    if len(lst) == 1:
        print(lst[0])
    elif len(lst) == 2:
        print(lst[0] + ' and ' + lst[1])
    else:
        print(', '.join(lst[:-1]) + ', and ' + lst[-1])

# Use it
smart_print(['A'])
smart_print(['A', 'B'])
smart_print(['A', 'B', 'C'])
smart_print(['A', 'B', 'C', 'D'])
```

This is much better. If you find an error in the code you only need to fix it in one place. If you think of a way to improve it, you only need to improve it in one place. And, as a bonus, the function name itself makes your code more self-documenting - each time you invoke the function its name makes it clear what you are doing, with no need for any comments. Brilliant!

Defining and using functions in this way is an example of **code modularisation**. It is an important and powerful technique, and is one of the cornerstones of good programming. You'll learn how to do it this week.

Defining functions

So far you've been using functions that are built-in to Python, such as `input()` , `print()`, and `len()`. Like many other languages, Python allows you to define your own functions.

Defining a function

You can define a function using a `def` statement, which has the following form:

```
def <name>():
    <statements>
```

For example:

```
def say_hello():
    print('Hello')
```

Once you've defined the function you can call it like any other function:

```
def say_hello():
    print('Hello')

say_hello()
```

Notice that the code inside the body of the function is not executed when the function is defined - it is only executed when the function is called.

Also notice that you must define the function before calling it:

```
say_hello() # Error - the function has not yet been defined
def say_hello():
    print('Hello')
```

Naming functions

The rules for naming functions are the same as for naming variables.

Although it is not required, it has become conventional to use snake case - lower case words, separated by underscores, for example `say_hello`.

You should choose names that help to document your code - naming the above function `say_hello`, for example, is more explanatory than naming it `hello`, or, even worse, `my_func`.

Adding parameters

You can add **parameters** to a function, to specify that the function should receive one or more **arguments** when it is called. You do this by adding parameter names in the brackets after the function name:

```
def say_hello(name): # Add a parameter called "name"
    print('Hello,', name)

say_hello('James') # Provide 'James' as an argument
```

When you call a function you must supply it with the right number of arguments - one for each parameter. If the function has no parameters then you must supply no arguments; if the function has one parameter then you must supply exactly one argument; and so on. If you supply the wrong number of arguments then Python will raise an error. The `say_hello()` function defined above has one parameter, so you must supply it with exactly one argument:

```
def say_hello(name):
    print('Hello,', name)

say_hello() # Error - not enough arguments
```

```
def say_hello(name):
    print('Hello,', name)

say_hello('James', 'Sarah') # Error - too many arguments
```

Adding default values

You can give parameters default values. If no argument is provided for that parameter then the function will use the default value.

```
def say_hello(name = 'James'): # Give the parameter a default value
    print('Hello,', name)

say_hello() # The default value will be used
say_hello('Sarah') # 'Sarah' will be used instead
```

You can use this feature to make arguments *optional* - Python won't raise an error if the argument is not supplied, it will just use the default value.

If you set the default value to `None` then you can use this to detect whether an argument was supplied for the parameter:

```
def say_hello(name = None):
    if name is None:
        print('No name was provided')
    else:
        print('Hello,', name)

say_hello()
say_hello('Sarah')
```

Returning values

A function always returns a value. By default it will return the object `None`, but you can use a `return` statement to get it to return whatever value you want.

```
def sum(x, y):
    return x + y # Specify a return value

print(sum(1, 2))
```

The function will exit immediately after a `return` statement, so any further statements in the function body will not be executed.

```
def sum(x, y):
    return x + y # The function exits here
    print('This will not be printed') # Not executed

print(sum(1, 2))
```

You can have multiple `return` statements (but only one will get executed):

```
def grade(mark):
    if mark >= 50:
        return 'Pass'
    else:
        return 'Fail' # Only one of these return statements will be executed

print(grade(73))
print(grade(35))
```

A function can only return one value. This value can, however, be a collection - a list, or a tuple, or a set, or a dictionary. It is fairly common to return a tuple. Here's an example in which a tuple with two elements is returned:

```
def ends(string):
    first_char = string[0]
    last_char = string[-1]
    # Return a tuple with two elements
    # Note that only the comma is needed - round brackets are assumed
    return first_char, last_char

print(ends('Australia'))
print(ends('Australia')[0])
print(ends('Australia')[1])
```

You might hear people describe this as a case in which the function returns two values. This is loose talk - the function returns just one value - a tuple. It's important to be aware of what's really going on here.

Supplying arguments by keyword

If your function has many parameters then it can be difficult to remember the order in which they need to be supplied when you call it, and it can be difficult for someone reading the code to tell which is which:

```
def divide(numerator, denominator, num_places):
    return round(numerator/denominator, num_places)

# Not clear to the reader which argument goes with which paramater
# Which one is the numerator? Which is the denominator?
print(divide(5, 6, 3))
```

To make things easier, arguments can be supplied by **keyword**:

```
def divide(numerator, denominator, num_places):
    return round(numerator/denominator, num_places)

# Now it's clear which argument goes with which paramater
print(divide(numerator=5, denominator=6, num_places=3))

# And we don't have to worry about their order
print(divide(num_places=3, denominator=6, numerator=5)) # Same result
```

Allowing an arbitrary number of arguments

Suppose you want a function that returns the smallest of some given numbers. Suppose you want to allow any number of numbers to be given.

One way would be to use a single parameter which expects a list. On this approach, you must provide exactly one argument, a list, but the list can be as long as you like:

```
def smallest(numbers): # Expects numbers to be a list
    smallest = numbers[0]
    for n in numbers:
        if n < smallest:
            smallest = n
    return smallest

print(smallest([42, 12])) # Provide a single list of 2 numbers
print(smallest([4, 11, 15, 2, 3])) # Provide a single list of 5 numbers
```

Another way is to precede the name of the parameter with an asterisk. Then you can provide as many numbers as you like, not together in a single list, but each as a separate argument in its own right. The function will automatically combine them into a tuple:

```
def smallest(*numbers): # Add an asterisk to the front of the name
    smallest = numbers[0]
    for n in numbers:
        if n < smallest:
            smallest = n
    return smallest

print(smallest(42, 12)) # Provide 2 arguments
print(smallest(4, 11, 15, 2, 3)) # Provide 5 arguments
```

You can also precede the name of the parameter with two asterisks. This allows you to provide as many keyword arguments as you like. This time they are gathered into a dictionary.

```
def full_name(**names):  # Add two asterisks to the front of the name
    result = ''
    if 'first' in names: result = result + ' ' + names['first']
    if 'middle' in names: result = result + ' ' + names['middle']
    if 'last' in names: result = result + ' ' + names['last']
    return result.strip()

print(full_name(first='John', last='Smith'))
print(full_name(first='John', middle='Hubert', last='Smith'))
print(full_name(last='Smith', middle='Hubert', first='John'))
```

You can use both kinds of parameter, but if you do then the * parameter must precede the ** parameter.

Pass

The body of a function definition cannot be empty, so if you want a function definition with no content then you need to use the pass statement to avoid Python raising an error.

```
def my_func():
    return None
```

Alternatively, you could put an expression in the body, such as `{}`, or get the function to return `None`,

Functions are given objects

It's important to be aware that when you pass an argument to a function, that function has direct access to the object and can change the object (if its mutable). Here's an example:

```
def add10(x):
    x.append(10)

lst = [1, 2, 3]
add10(lst)
print(lst)
```

Notice that the function defined here changes the object that is passed to it as an argument. In this case we say that the function has **side effects** - it affects the state of the program outside the function.

One way to avoid the original list being changed is to use **list** method `copy()` as shown below:

```
def add10(x):
    x.append(10)

lst = [1, 2, 3]
lst1 = lst.copy() # create a copy of the object
add10(lst1)
print(lst1)
print(lst) # will not be modified
```

Please note that the above will not apply to parameters passed by **value** (instead of **reference** as the list example above) as shown below:

```
def add10(x):
    x = x + 10
    print(x) # outputs 15

num = 5
add10(num)
print(num) # outputs 5 NOT 15
```

Variable scope

If you create a variable inside a function then that variable is only defined inside the function. We say that the variable's **scope** is limited to the function, or that the variable is **locally defined**. If you try to use a variable outside its scope then Python will raise an error.

```
def my_func():
    x = 5

print(x) # Error - x is only defined inside the function
```

Even if you have used the same variable name outside the function, changes to variables defined inside the function are limited to occurring inside the function. This can be a problem if you use locally defined variables with the same name as globally defined variables (i.e. variables not declared within the scope of a function) - this is known as **variable shadowing**.

```
x = 3 # Globally defined x

def my_func():
    x = 5 # Locally defined x, no change to the globally defined x

my_func() # No change to the globally defined x
print(x) # Prints 3, not 5
```

If you want to use globally defined variables inside functions, the safest approach is to provide them to the function as arguments.

```
x = 3

def my_func(y):
    return y + 2 # Add 2 to the number provided and return the result

x = my_func(x) # Assign to x the result of of my_func(x) - changes x
print(x) # Prints 5
```

Nested functions

You can define a function inside another function. When you do, the inside function is called a **nested function**. Here's an example:

```
def acronym(string):
    result = ''
    words = string.split(' ')
    def upper_first(string): # A nested function
        return string[0].upper()
    for word in words:
        result += upper_first(word)
    return result

print(acronym('World Health Organisation'))
```

Because a nested function is defined inside an enclosing function, it is only available to be called inside that enclosing function. The following program generates an error, because the nested function is called outside its enclosing function:

```
def acronym(string):
    result = ''
    words = string.split(' ')
    def upper_first(string): # Only available inside acronym
        return string[0].upper()
    for word in words:
        result += upper_first(word)
    return result

print(upper_first('hello')) # Error - upper_first is not available here
```

Lambda functions

You can refer to a function without giving it a name.

Suppose, for example, you have a list of names and you want to sort those names by their *last* letter. You can use the list's `sort()` method to do this. By default, `sort()` sorts them alphabetically, but you can override this default by providing a function to use as the sorting key.

If you like, you can first define the function, giving it a name, and then provide it by name to `sort()`:

```
names = ['Geoff', 'Kim', 'Louise', 'Tam', 'Helen']
def last_letter(name):
    return name[-1]
names.sort(key = last_letter) # Use the function defined above
print(names)
```

But you don't need to. You can refer to the function directly when you call `sort()`, without giving it a name. You do this by using a **lambda function**:

```
names = ['Geoff', 'Kim', 'Louise', 'Tam', 'Helen']
names.sort(key = lambda name: name[-1]) # Use a lambda function
print(names)
```

A lambda function is an **expression** (not a statement) whose value is a function. You can think of a lambda function as being a **function literal**.

The syntax of a lambda function is as follows:

```
lambda <parameters>: <expression>
```

Note that there is no `return` in a lambda function.

Lambda functions can have more than one parameter. Here's a lambda function with two parameters:

```
lambda a, b: a + b
```

You can use a lambda function just like you use function names. You can call the function it refers to by using the usual round brackets notation (note that you typically need to put parentheses around the lambda function when you call it, to avoid confusion with neighbouring code):

```
print((lambda a, b: a + b)(2, 4))
```

And you can use it to assign a value to a variable:

```
f = lambda a, b: a + b
```

```
print(f(2, 4))
```

Note what is going on in this last example. We are using the lambda function `lambda a, b: a + b` to assign a value to a variable `f`. The value of the lambda function is a function - you can think of it as a literal for that function. So `f` is being assigned a function. We can then use `f` like any other function name. In line 2, we call the function, using `f(2, 4)`.

Compare the above with the following example:

```
def f(a, b):
    return a + b
print(f(2, 4))
```

The two examples are similar, but there are some subtle differences. In both examples we end up with `f` being the name of a function. But we get there in two different ways. In the first example, we <u>assign</u> `f` the function using an assignment statement and a lambda function that refers to the function. In the second example, we <u>define</u> `f` using a `def` statement that defines the function.

Functions are objects

Functions are objects, and you can use them in the same way you use other objects, such as numbers, strings, lists, and so on. Just as you might set a variable's value to a number, such as `1`, you might also set it to a function, such as `len()`. Also:

- You can assign a function to a variable
- A function can be an attribute of an object
- A function can be an element of a collection
- Functions can be keys in a dictionary
- You can pass a function as an argument of a function call
- You can return a function as the result of a function call

And so on. Because of this we say that Python functions are **first class**.

Because you can supply functions as arguments to functions, you can create functions that operate on functions:

```
def add(x, y):
    return x + y
def subtract(x, y):
        return x - y
def apply(f, x, y): # This function applies function f to values x and y
        return f(x, y)

print(apply(add, 10, 1))
print(apply(subtract, 10, 1))
```

Here's another example. In this case we define a function `compose`, which takes two functions `f` and `g` as arguments and returns a function - the *composition* of `f` and `g`, which is the function that takes an argument `x` and returns `f(g(x))`:

```
def add1(x):
    return x + 1
def subtract1(x):
        return x - 1
def compose(f, g): # This function returns a function
        return lambda x: f(g(x))

add2 = compose(add1, add1) # add2 is a new function
print(add2(10))
do_nothing = compose(add1, subtract1) # do_nothing is a new function
print(do_nothing(10))
```

Functions that take functions as arguments, or return functions as values, are known as **higher order**

Generators

Before we leave the topic of functions, there is one special type of function that you should know about.

Suppose you have a function that returns a collection of objects, perhaps a list. Suppose it is the following one:

```
def squares():
    result = []
    for x in range(10):
        result.append(x**2)
    return(result)

print(squares())
```

The function returns the full list, which you can then **iterate** over:

```
def squares():
    result = []
    for x in range(10):
        result.append(x**2)
    return(result)

for x in squares():
    print(x)
```

Rather than getting the function to return the full list, you can get it to return the elements one at a time, by using a `yield` statement instead of a `return` statement:

```
def squares():
    for x in range(10):
        yield x**2 # Use a yield statement

for x in squares():
    print(x)
```

Notice what happens when you print:

```
def squares():
    for x in range(10):
        yield x**2

print(squares())
```

The function now returns a special kind of object, called a **generator** - it does not return the full list. This generator object generates the elements as they are needed.

Why would you do this, rather than have the function return the full list at the outset? If the list is large, and if you don't need its elements all at once, then it is a good way to save memory.

Generator expressions

There is an even more concise way to make a generator. Rather than defining a function that returns a generator, you can use a **generator expression**. It is exactly like a comprehension, but you use round brackets. This is why there is no tuple comprehension - the round brackets are used for generator expressions instead.

```
squares = (x**2 for x in range(10)) # Get a generator from a generator expression

print(squares)
for x in squares:
    print(x)
```

Further reading

You might find the following helpful:

- The Python Tutorial at w3schools.com