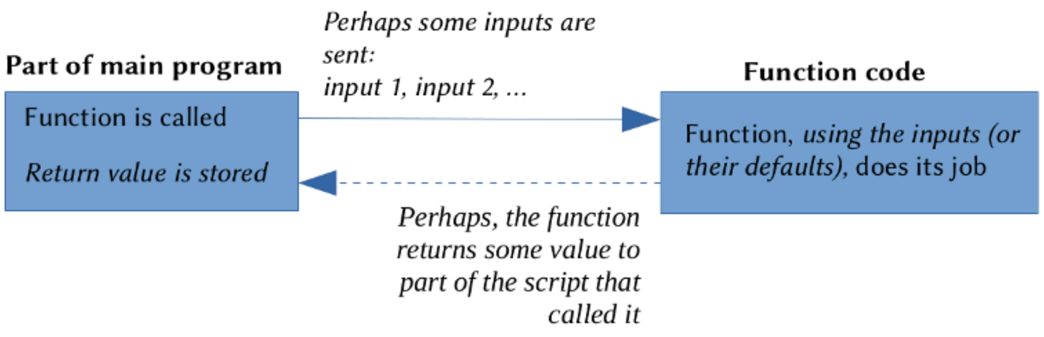
Functions

What is a function?

Put simply, a function is a collection of commands that you might want to reuse. It can take input and produce output (though it doesn't have to do either).


Part of main program
Function is called
Return value is stored
Perhaps some inputs are sent: input 1, input 2, ...
Function code
Function, *using the inputs (or their defaults),* does its job
Perhaps, the function returns some value to part of the script that called it

Built-in functions

You've already used some built-in functions. For example:

- lots of the SymPy commands we've looked at are functions, for example when you write `diff(x**2,x)` you are calling the function `diff` and giving it a mathematical expression and telling it the independent variable you want it to differentiate with respect to.
- `print` is a function that takes as input a string and outputs this string followed by a new line character `\n` to the Python shell. It does not return anything, meaning that you don't store a return variable by writing something like `a = print("Hello")`.

  You can send `print` a second parameter `end=""` to tell it what to put at the end of the string instead of a new line character. (To see full details, go to the Python shell and enter `help(print)`.) For example, you can change the character that Python puts at the end of the string by writing something like `print("Hello", end=",")` to put `","` at the end of what you print. However, `end` defaults to `\n`, so if you don't specify it then `\n` is what is used.

Defining your own functions

It is sometimes useful to define your own function. This gives you a command you can run from elsewhere in your program, perhaps multiple times.

Here is a simple definition of a function. It takes an integer and prints it out. You can place this anywhere in your script before the first time you want to call the function. Note that the code inside the function definition is indented (like we saw with `if` and `for`).

```
def printme(i):
    print("You asked me to print out this: {}".format(i))
```

On its own, the function doesn't do anything. All the code above does is defines the function, it doesn't tell it to run. You get the function to act by *calling* it using the name you gave it (in this case, `printme`). The good thing about having code in a function is that you can call it multiple times in your script, as in the code below.

```
def printme(i):
    print("You asked me to print out this: {}".format(i))

printme(7)
printme("hello")
printme(5*9)
```

The function we wrote, `printme` takes a variable as input and doesn't return anything. The action it takes is to print out something with some extra text.

Below is a function that takes an input *and* has a return value - it returns double what you give it as input using the command `return`. Here, we call the function we wrote, `doubleme`, three times, and collect the return values into variables `a`, `b` and `c`. Then we print the values of `a`, `b` and `c` as a tuple.

```
def doubleme(number):
    return 2*number

a = doubleme(2)
b = doubleme(57)
c = doubleme(-5)

print((a,b,c))
```

Of course, we are writing this function to expect a number. Try calling `doubleme` with other input types like a string or a list and see what happens. You should get some curious results, which may teach you something about the `*` operator in Python.

Here is a more complicated function, which I have called `is_square`. Can you see what this is doing? Can you see what the loop is doing? Run it and see if you are right.

```
def is_square(n):
    if n<1:
        return False
    else:
        for i in range(int(n/2)+1):
            if i*i == n:
                return True
        return False

for i in range(0,100):
    if is_square(i):
        print(i)
print("That's all, folks.")
```

Context

In the example above, which defined and used the function `is_square`, the function definition was at the top of the file and the function was called (used) from code below it. If the function is called before it is defined, Python is not happy. Try this code to see what happens.

```
for i in range(0,100):
    if is_seven(i):
        print(i)
print("That's all, folks.")

def is_seven(n):
    if n==7:
        return True
    else:
        return False
```

As well as defining functions before they are called, it is important to think about what variables are used within the function. For example, the function `what_outcome` below defines a variable `outputstr` to hold the output string. Run the code below - it should produce some output and then cause an error. Can you see why?

```
def what_outcome(input):
    if input == 11:
        outputstr = "Houston, Tranquillity Base here. The Eagle has landed."
    elif input == 13:
        outputstr = "Okay, Houston, we've had a problem here."
    print(outputstr) # this one prints

what_outcome(11)
print(outputstr) # this one causes an error
```

Here, the variable `outputstr` is defined within the function and is therefore a *local* variable. It is not available to the main program code outside the function.

Now consider the following code. What would you expect to happen? What actually happens?

```
outputstr=""
def what_outcome(input):
    if input == 11:
        outputstr = "Houston, Tranquillity Base here. The Eagle has landed."
    elif input == 13:
        outputstr = "Okay, Houston, we've had a problem here."

what_outcome(11)
print(outputstr)
```

Here, there are actually *two* variables called `outputstr`. At the top of the code is defined a *global* variable called `outputstr`. This is global in the sense that it is used by the script as a whole. Then within the function a *local* variable called `outputstr` is defined that is local in the sense that it exists only within the `what_outcome` function. It is this local variable that is used to store one of the outcome messages. At the bottom, the code attempts to print `outputstr` and, because we are outside the function at that point, it is the (blank) global variable that is printed. Make sure you understand this, and ask for help if you need it. This can be a difficult concept to get your head around, but it can cause some subtle and hard-to-detect errors if you don't understand it.

There are two ways to change a global variable with code that is inside a function.

1. You can have the function return its value for the global variable and store this in the main program. This is demonstrated in the code below.

```
outputstr=""
def what_outcome(input):
    if input == 11:
        outputstr = "Houston, Tranquillity Base here. The Eagle has landed
    elif input == 13:
        outputstr = "Okay, Houston, we've had a problem here."
    return outputstr

outputstr = what_outcome(11)
print(outputstr)
```

2. You can ask the function to access the global variable rather than creating a local version using the command `global`.

```
outputstr=""
def what_outcome(input):
    global outputstr
    if input == 11:
        outputstr = "Houston, Tranquillity Base here. The Eagle has landed
    elif input == 13:
        outputstr = "Okay, Houston, we've had a problem here."

what_outcome(11)
print(outputstr)
```

Documentation

Previously, we saw that documentation could be added to the top of a file. A documentation string can also be added to the top of a function, so that people can ask for `help()` on the function specifically.

```
"""
This docstring applies to the whole file
"""

def say_hello():
    """
    This docstring applies specifically to the say_hello() function.
    """

    print("Hello")
```

Exercises

1. Write a function that takes no input and has no return value, that simply prints out a message. Call this several times.

2. Write a function that takes as input a string and prints a message that says "Here is your string: " and the string. Call this several times with different inputs

3. Write a function that takes as input a number and returns seven times that number. Write main program code that calls the function and prints out the returned value.
4. `abs(x)` is a function that returns the absolute value of a number `x`. Set a variable to an arbitrary number (positive or negative), and use `abs` to print out its absolute value. Try integers and floats (decimals).
5. `bin(x)` is a function that converts an integer `x` to binary. Use it to write a program that converts your house or flat number to binary (or some other number you think of).
6. `len(s)` is a function that returns the length (the number of items) in an object. Create a list and use `len` to print out its length. Create a string and use `len` to print out its length.
7. `max(k)` is a function that returns the largest item in a list `k`. Make a list of numbers and use `max` to find the biggest.
8. `max` can also be used to find the largest of several arguments, as `max(a,b,...)`. Make several integer or float variables and use `max` to find the largest of them.
9. `min` works in the same two ways as `max`, but does something different. Can you guess what? Have a go and see if you are right!
10. `round(n,d)` returns float `n` rounded to `d` digits after the decimal point. If `d` is omitted, it rounds to the nearest integer. Set a float `n` and use `round` to round this to seven decimal places.