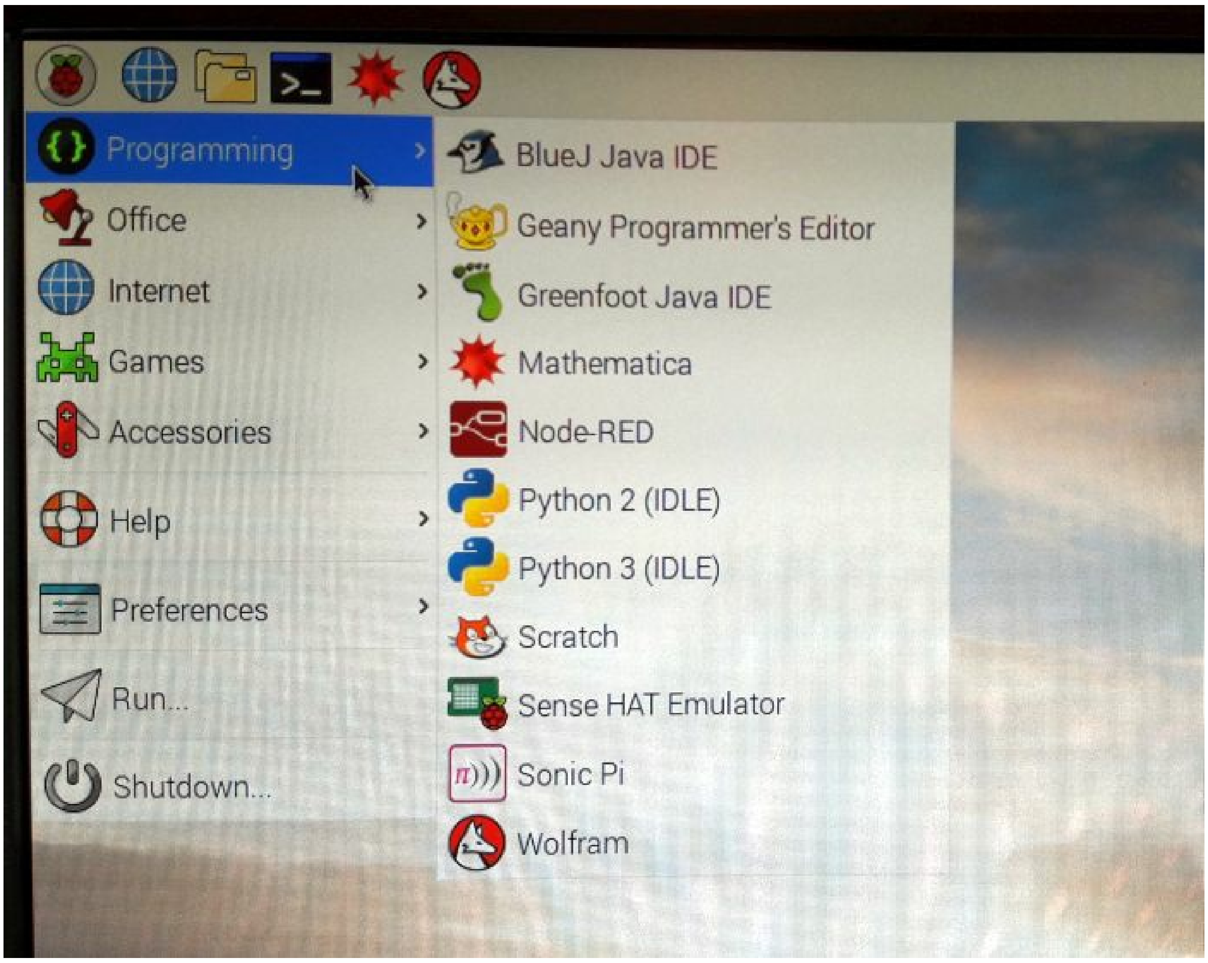
Introduction to Python

- Raspbian Integrated development environment (IDE)
- Basic Practise
- Data Types
- Scripts
- Functions

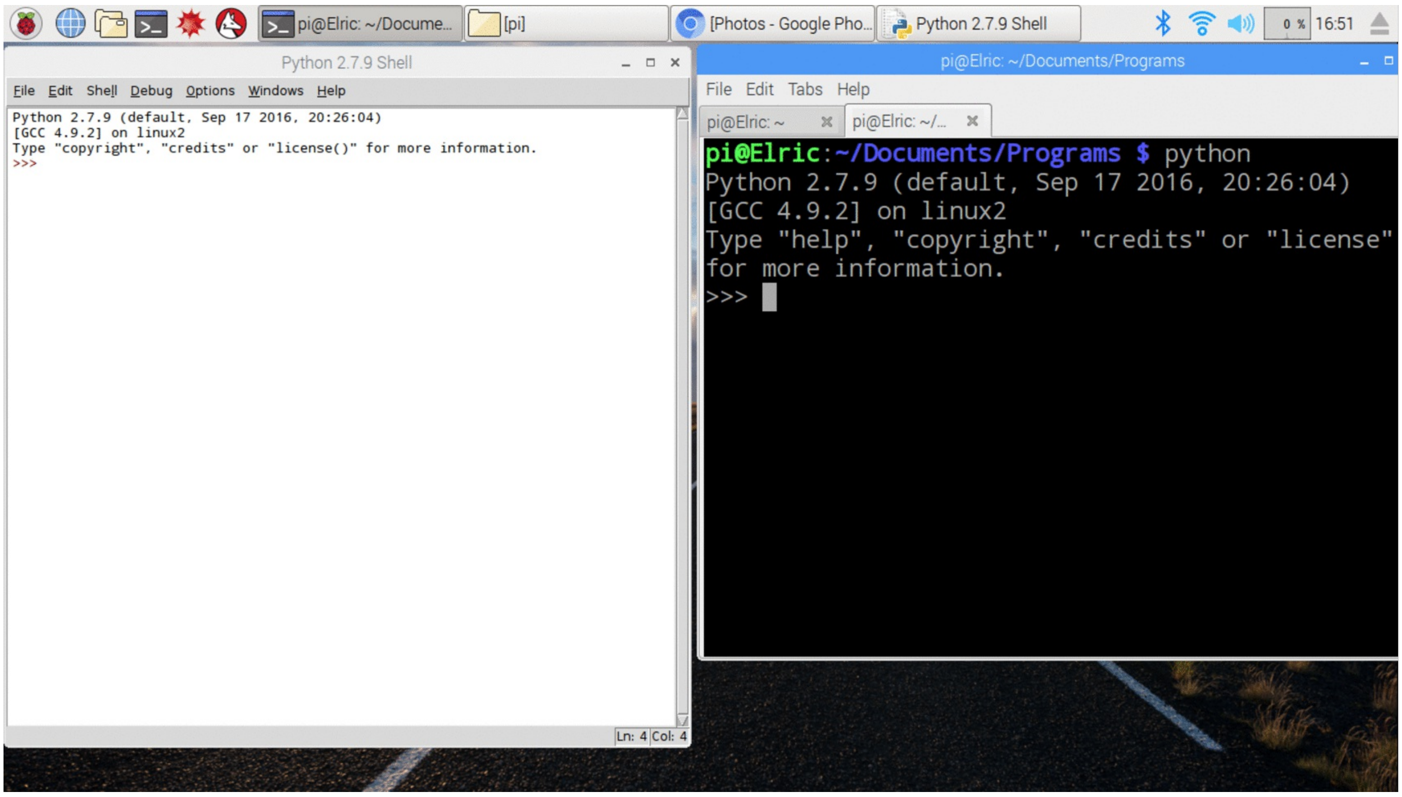
Raspbian Integrated development environment (IDE)

Using a Raspberry Pi is an excellent way of getting into programming, especially if you are a newbie and want to learn about programming in different programming languages. To start getting into programming, then the first thing to do is to get yourself an IDE (Integrated Development Environment). You need to write code to make your RPi do things, and an **IDE is a tool to write, test and run code**. The RPi supports lots of different languages to write your code, so there is a wide choice (look at the screenshot bellow).



For our course, we are going to use Python as our main programming language. It has a tidy syntax, easy to understand.

In the menu, we have access to the Python interpreters. Also, we can use the Geany Programmer's Editor as an IDE. Also, we can use editors available in the terminal like vim, nano, etc. We can also get used of the Python interpreter from the terminal and run scripts (see bellow).



```
Python 2.7.9 (default, Sep 17 2016, 20:26:04)
[GCC 4.9.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>>
```

```
pi@Elric:~/Documents/Programs $ python
Python 2.7.9 (default, Sep 17 2016, 20:26:04)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license"
for more information.
>>>
```

About Python Language
Remember that you are intelligent, and you can learn, but the computer is simple and very fast, but can not learn by itself. Therefore, for you to communicate instructions on the computer, it is easier for you to learn a computer Language (e.g. Python) than for the computer to learn English.

Python can be **easy to pick up and friendly to learn**. Python is a **general-purpose** interpreted , interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. There are two main python versions: 2.7 and 3. For this course, we will use 2.7 since it is the most common or popular used.

Basic Practise

Let's get familiar with Python by playing in the terminal in the interactive mode (you type a line at a time, and the interpreter responds). You invoke the interpreter and brings up the following prompt:

```
$python
Python 2.7.9 (default, Sep 17 2016, 20:26:04)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Strings, integers, and floating points:

```
>>> print "Hello, Python!"
>>> x = 1          # Integer assignment
>>> y = 1005.00    # Floating points
>>> name = "John" # A string
>>> print x
>>> print y
>>> print name
```

We can perform mathematical calculations in Python using the basic operators +, -, /, *, %.

```
>>> 4 + 5
>>> 4 - 3
>>> 2 * 3
>>> 2**3
>>> 10 % 7 # Remainder of a division
>>> 3 / 4
```

Why we get zero? In Python 2.x, where integer divisions will truncate instead of becoming a floating point number. You should make one of them a float:

```
>>> float(3)/4
>>> 3 / float(4)
```

In order to deal with classic division in Python 2.x, we can import a module called future which import Python 3 functions into Python 3.

```
>>> from __future__ import division
>>> 3/4
```

// is the floor division in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero.

```
>>> 11.0 / 3
>>> 11.0 //3
>>> -11.0 // 3 # Result floored (rounded away from zero)
```

In Python, the standard order of operations are evaluated from left to right following order (memorised by many as PEMDAS):

Name	Syntax	Description
Parentheses	(...)	Happening before operating on anything else.
Exponents	**	An exponent is a simply short multiplication or division, it should be evaluated before them.
Multiplication and **D**ivision	* /	Multiplication is rapid addition and must happen first.
Addition and **S**ubtraction	+ -	

```
>>> 3/4 * 5  # First division and then Multiplication
>>> 3.0 / 4 * 5
>>> (3.0 / 4) * 4
>>> 2**8
>>> z = float(5)
>>> z
>>> z = int(5.25)
>>> z
>>> 10%7 # Remainder of a division
>>> 'abc' + 'fgb' # strings
```

Comparison operators:

Name	Syntax
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

```
>>> 2 == 3
False
# We got a boolean
>>> 3 == 3
True
>>> 2 < 3
True
>>> "a" < "aa"
True
```

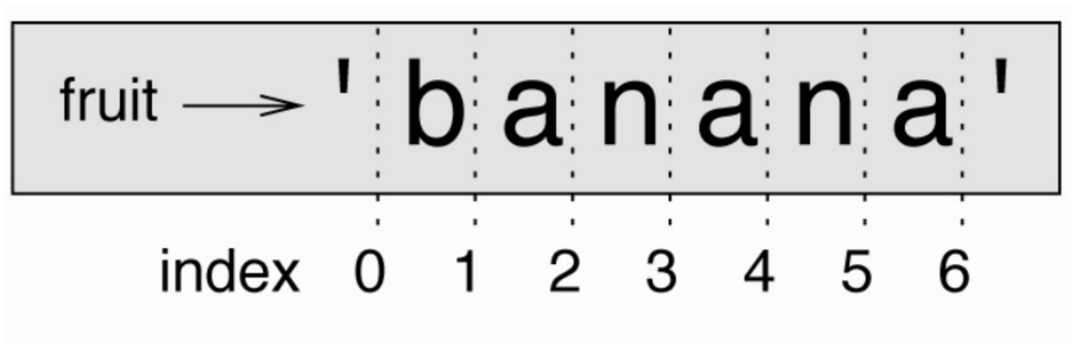
Data Types

The data stored in memory can be of different types; Python has five: **Numbers, Strings, List, Tuple, and Dictionary**.

```
>>> type(x) # numbers
>>> type(y)
>>> type(name) # String
```

Strings in Python are a set of characters represented by the quotation marks. Python allows for either pair of single or double quotes.

Since strings are a sequence of characters, we can use indexes to query a section of the sequence. Subsets of strings can be taken using the slice operator ([] and [:]) with indexes starting at 0 at the beginning of the string and working their way from -1 to the end.



The plus (+) sign is the string concatenation operator, and the asterisk (*) is the repetition operator. For example:

```
>>> string = 'Hello World!'
>>> print string          # Prints complete string
>>> print string[0]       # Prints first character of the string
>>> print string[2:5]     # Prints characters starting from 3rd to 5th
>>> print string[2:]      # Prints string starting from 3rd character
>>> print string * 2      # Prints string two times
>>> print string + "TEST" # Prints concatenated string
>>> # How to be careful with quotes
>>> 'You're using single quotes that will produce an error'
>>> "Now you're not going to have problems. No errors in the string!"
```

Lists are the most versatile data types in Python. A list contains items separated by commas and enclosed in square brackets ([])—similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([] and [:]) with indexes starting at 0 at the beginning of the list and working their way to ending -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.

```
>>> list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
>>> tinylist = [123, 'john']

>>> print list          # Prints complete list
>>> print list[0]       # Prints first element of the list
>>> print list[1:3]     # Prints elements starting from 2nd till 3rd
>>> print list[2:]      # Prints elements starting from 3rd element
>>> print tinylist * 2  # Prints list two times
>>> print list + tinylist # Prints concatenated lists
```

A **tuple** is another sequence data type that is similar to the list. It consists of some values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ([]), and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated—**immutable**. You would use tuples to present things that should not be changed, such as days of the week, or dates on a calendar. Tuples can be thought of as read-only lists.

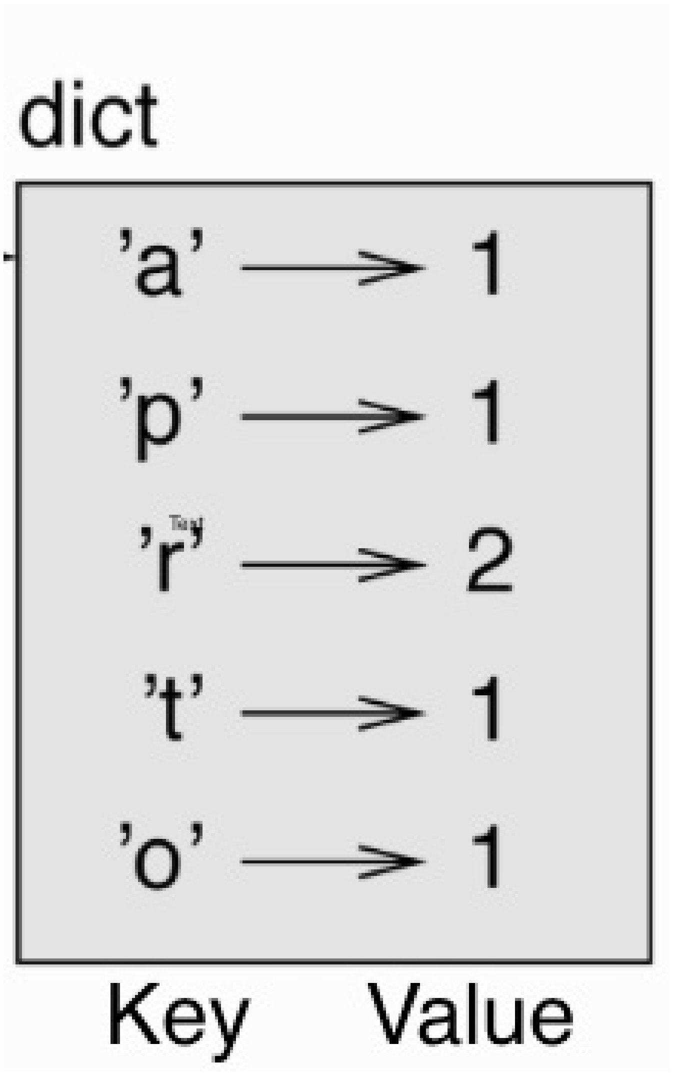
```
>>> tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')

>>> print tuple           # Prints complete list
>>> print tuple[0]          # Prints first element of the list
>>> print tuple[1:3]        # Prints elements starting from 2nd till 3rd
>>> print tuple[2:]         # Prints elements starting from 3rd element
>>> print tinytuple * 2     # Prints list two times
>>> print tuple + tinytuple # Prints concatenated lists
```

Invalid operations on a tuple but valid on a list:

```
>>> tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
>>> list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]
>>> tuple[2] = 1000     # Invalid syntax with tuple
>>> list[2] = 1000      # Valid syntax with list
```

Until now we have learnt about sequences, but now we will learn about mapping in Python. **Dictionaries** are kind of hash table type common in other languages. Mappings are a collection of objects that are stored by a key, unlike a sequence that stored objects by their relative position. In the case of mappings, they do not retain order since they have objects defined by a key. A dictionaries work like **associative arrays and consist of key-value pairs**. A key can be almost any Python type but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object. Dictionaries are enclosed by curly braces {} and values can be assigned and accessed using square braces [].



```
>>> dict = {}
>>> dict['one'] = "This is one"
>>> dict[2]     = "This is two"
# keys are: name, code and dept; values are: john, 6734 and sales
>>> tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

>>> print dict['one']       # Prints value for 'one' key
>>> print dict[2]           # Prints value for 2 key
>>> print tinydict          # Prints complete dictionary
>>> print tinydict.keys()   # Prints all the keys
>>> print tinydict.values() # Prints all the values
```

To quit the Python interpreter:

```
>>> quit()
```

Scripts

A Script is a sequence of statements (lines) into a file using a text editor and tells Python interpreter to execute the statements in the file.

- We can write a program in our script like a recipe or installation of software. At the end of the day, a program is a **sequence** of steps to be done in order.
- Some of the steps can be **conditional**, that means that sometimes they can be skipped.
- Sometimes a step or group of steps are to be **repeated**.
- Sometimes we store a set of steps that will be used over and over again in several parts of the program (**functions**).

Note: Have a look on the code style guide for a good coding practise. As a fist good practise, do not name files or folders with space in between: Auful! → example 1.py; Great! → **example_1.py, exampleOne.py, example_one.py**

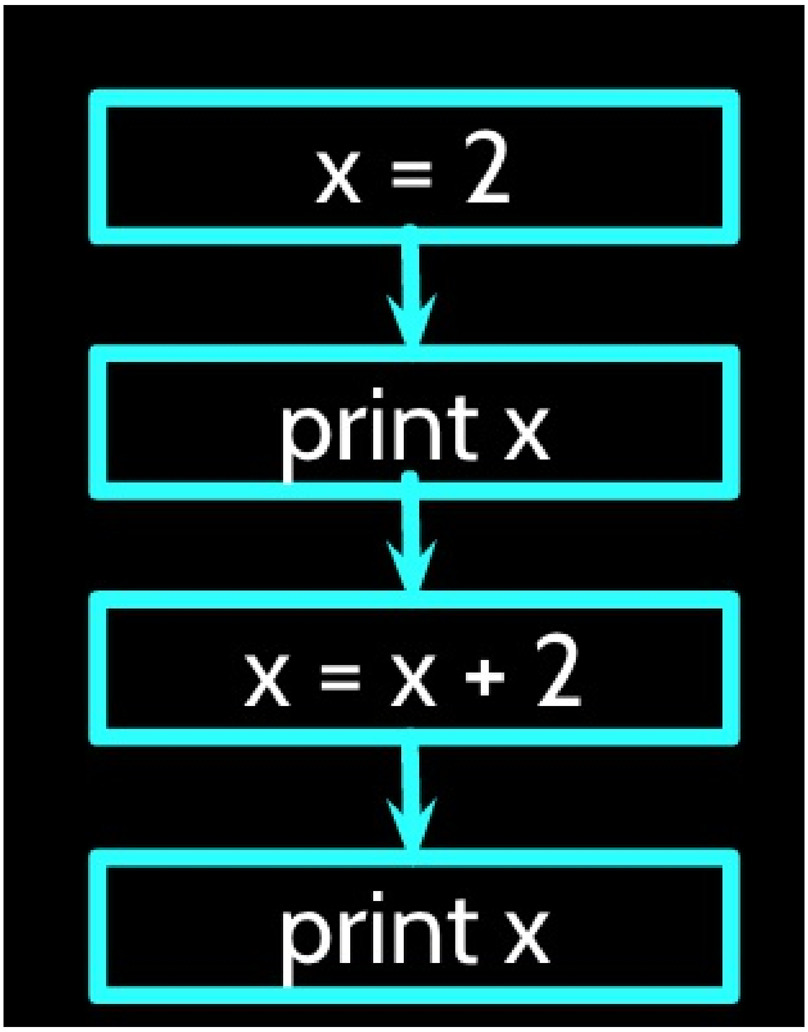
We will make a simple script:

```
$ pwd
$ /home/pi
$ mkdir codes/python_examples
$ cd codes/python_examples
$ nano example_fllow.py
```

Then you can type in the editor:

```
#!/usr/bin/env python
x = 2
print x
x = x + 2
print x
```

When a program is running, it flows from one step to the next. As programmers, we set up “paths” for the program to follow.



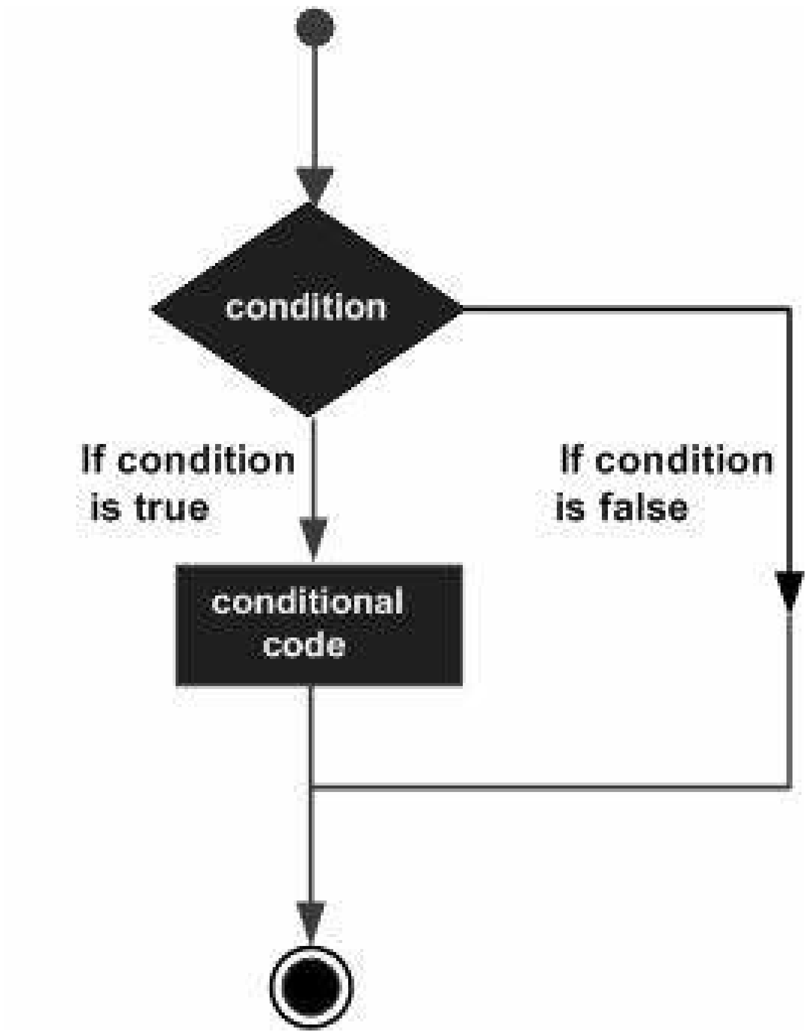
Close the text editor, and then you can execute it in two ways:

```
$ python example_fllow.py
```

The other is to give the script the access permissions to be an executable file through the chomod Linux command:

```
$ chmod u+x example_fllow.py
$ ./example_fllow.py
```

Now let's do an example where we have a **conditional** that implies a decision-making about a situation. Decision making is the anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions. The following diagram illustrates the conditional:



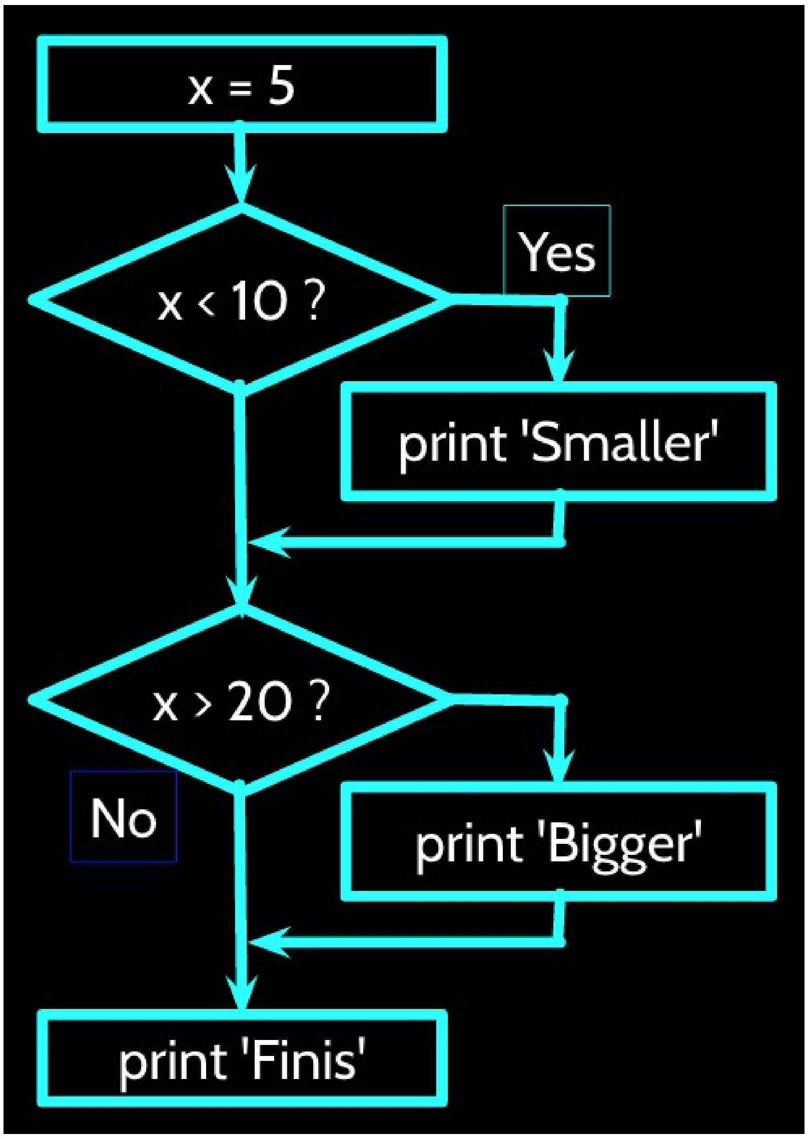
```
$ nano example_conditional.py
```

Now let's add the code:

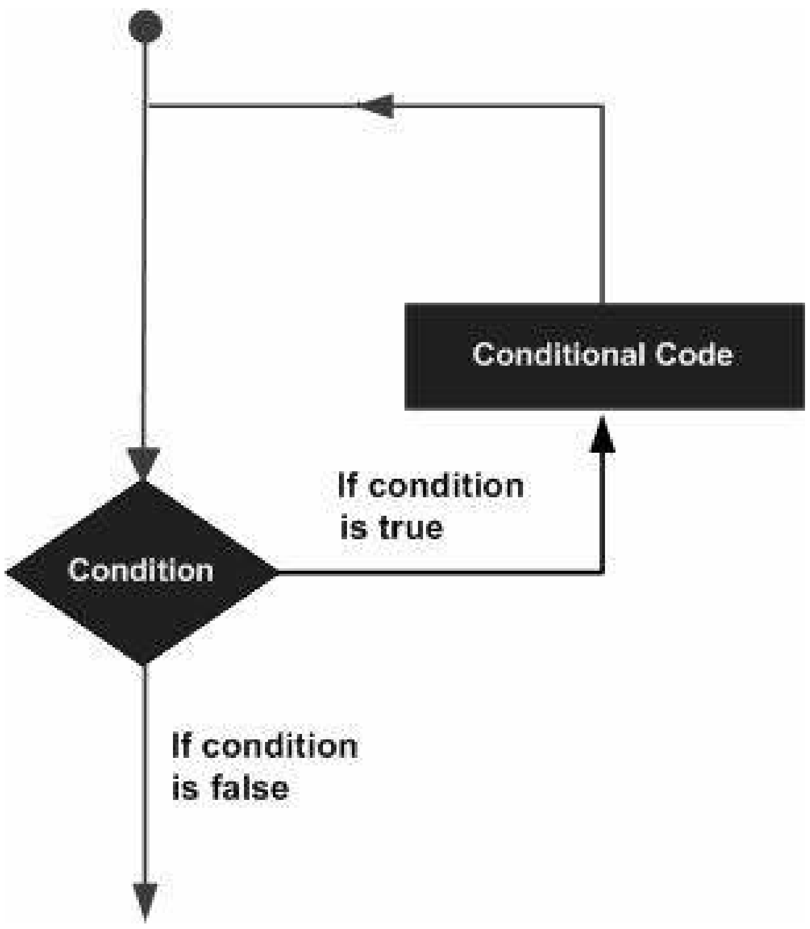
```
#!/usr/bin/env python
x = 5
if x < 10:
    print 'Smaller'
elif x > 20:
    print 'Bigger'
print 'Finis' #outside conditional
```

```
$ chmod u+x example_conditional.py
$ ./example_conditional.py
```

Flow of the code:



A **loop statement** allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement:



While loop repeats a statement or group of statements while a given condition is *TRUE*. It tests the condition before executing the loop body.

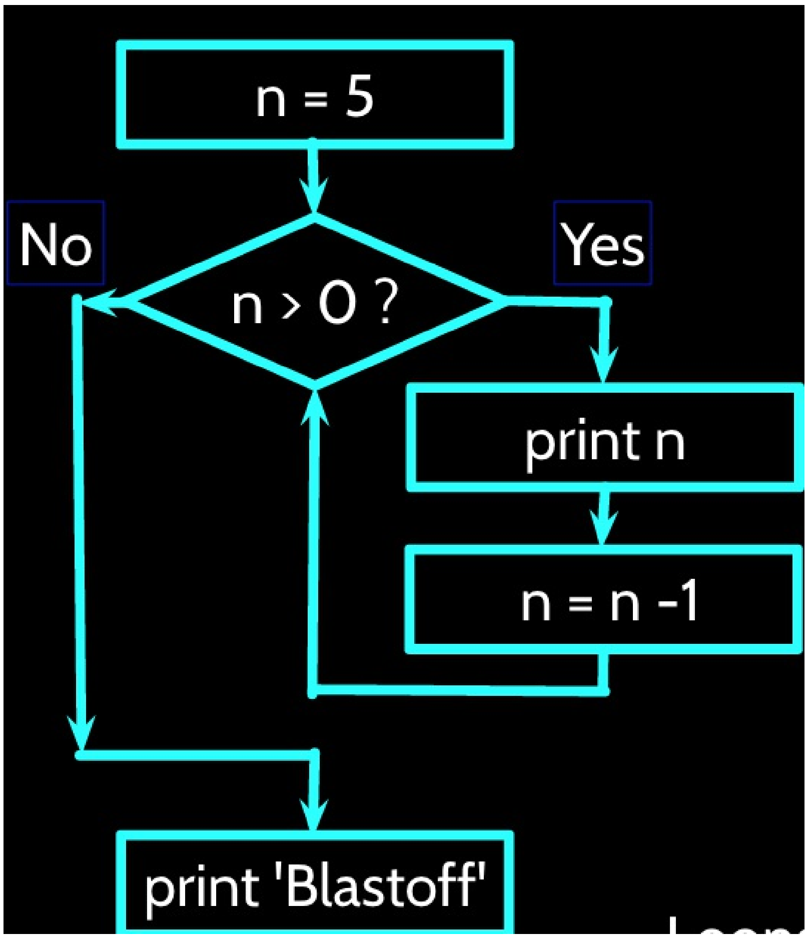
Now let's add the code to our script called *example_while_loop.py*:

```
#!/usr/bin/env python
n = 5
while n > 0:
    print n
    n = n - 1
print 'Blastoff!' #outside loop
```

Before running, remember to give the permissions:

```
$ chmod u+x example_while_loop.py
$ ./example_while_loop.py
```

Flow of the code:



Loops (repeated steps) ha‰ve *iteration variables* that change each time through a loop (like *n*). Often these *iteration variables* go through a sequence of numbers.

For loop executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

Now let's add the code to our script called *example_for_loop.py*:

```
#!/usr/bin/env python

# Area of a circle = pi * r**2

# Library
import numpy as np

# List are called interables
list = [1, 2, 3, 4, 5, 6]

for radius in list:
    area = np.pi * radius ** 2
    print "The area of a circle of radius ", radius
    print "cm is", area, "cm^2"
print "Finished to calculate the areas of circles"
```

```
$ chmod u+x example_for_loop.py
$ ./example_for_loop.py
```

Here we are importing the Numpy library. That is the fundamental package for scientific computing with Python. We are adding a short alias to the library to call its methods, in this case, the value of Pi.

Functions

A function is a block of organised, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

Now, let's make a function that can be used in the for loop example.

```
$ nano example_function_circle_area.py
```

```python
#!/usr/bin/env python

# Area of a circle = pi * r**2

# Library Numpy
import numpy as np


def area_circle(radius):
    'Function that calculates the area of a circle'
    area = np.pi * radius ** 2
    return area

# List are called interables
list = [1, 2, 3, 4, 5, 6]

for radius in list:
    area = area_circle(radius)
    print "The area of a circle of radius ", radius
    print "cm is", area, "cm^2"
print "Finished to calculate the areas of circles"
```

```
$ chmod u+x example_function_circle_area.py
$./example_function_circle_area.py
```

We can see that we get the same result but it is more organise and we can use the function in other section of our code.

Now let's ask the user to provide a list:

```
$ nano example_function_circle_area_user_1.py
```

```python
# Area of a circle = pi * r**2

# Library Numpy
import numpy as np
# Library to Safely evaluate an expression node
# or a string containing a Python expression
import ast

# List are called interables
list_raw = raw_input('Provide a list of radius in cm like \
[3, 2, 12, 6]: \n')
list = ast.literal_eval(list_raw)


def area_circle(radius):
    'Function that calculates the area of a circle'
    area = np.pi * radius ** 2
    return area


for radius in list:
    area = area_circle(radius)
    print "The area of a circle of radius ", radius
    print "cm is", area, "cm^2"
print "Finished to calculate the areas of circles"
```

```
$ chmod u+x example_function_circle_area_user_1.py
$./example_function_circle_area_user_1.py
```

If we do not use the ast library to evaluate a string containing a Python expression (in this case a list), we will get an error since Python will interpret as a string type and not a list type.

A second way to do it is by using the sys module which provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.

Now let's ask the user to provide a list by passing the strings directly:

```
$ nano example_function_circle_area_user_2.py
```

```
#!/usr/bin/env python

# Usage instructions:
# ./example_function_circle_area_user_2.py "[1, 2, 3]"

# Area of a circle = pi * r**2

# Library Numpy
import numpy as np
# Library to Safely evaluate an expression node
# or a string containing a Python expression
import ast
# Module provides access to some variables
# used or maintained by the interpreter
import sys


list_raw = sys.argv[1]
list = ast.literal_eval(list_raw)


def area_circle(radius):
    'Function that calculates the area of a circle'
    area = np.pi * radius ** 2
    return area
```

```
for radius in list:
    area = area_circle(radius)
    print "The area of a circle of radius ", radius
    print "cm is", area, "cm^2"
print "Finished to calculate the areas of circles"
```

```
$ chmod u+x example_function_circle_area_user_1.py
$./example_function_circle_area_user_2.py "[1, 2, 3]"
```

References [1, Charles Severance course: Python for everybody]