Introduction to Python

Programming in Python using our Raspberry Pi

Connect your raspberry pi and log in (ssh into it).

Note: You can follow this introduction to Python also in your laptop. If it's a Mac, simply open the terminal and follow this text. If it's a Windows computer, you need first to install Python, then open a powershell and follow this text. In some Windows machines, Python version 2.7 doesn't seem to install correctly. If you have problems, uninstall it and install Python version 3.6 or above.

The Python Interpreter

In the command-line interface (cli) of your Pi, type

```
pi@rpi3dragon1:~ $ python
```

You will see something similar to

```
Python 2.7.13 (default, Apr 28 2017, 15:56:03)
[GCC 4.8.4] on netbsd7
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Notice the *version of the Python interpreter* that you are using. In this example it is version 2.7 (technically, 2.7.13 but we can safely ignore the last version number in this context).

The help command of the python interpreter provides access to tons of useful explanations on different topics related to both, the Python interpreter and Python as a programming language itself.

```
>>> help()

Welcome to Python 2.7!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/2.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".
```

```
help> keywords

Here is a list of the Python keywords.  Enter any keyword to get more help.

and                 elif                if                  print
as                  else                import              raise
assert              except              in                  return
break               exec                is                  try
class               finally             lambda              while
continue            for                 not                 with
def                 from                or                  yield
del                 global              pass

help>
>>>
```

The Python prompt

The three `>` signs form what is called *a prompt*: The computer is waiting for you to type in something.

Type `2+3` and press enter. You should see this:

```
>>> 2+3
5
>>> 2-3
-1
```

That is, the python interpreter prints out the result of adding 2 and 3.

Additions, Multiplications and Powers

More examples (try them out and check that they do work in your RPi):

```
>>> 2*3         # Multiplications and additions/subtractions
6
>>> 2*7+4
18
>>> 2*7+4*(4-1)
26
>>> 2**1        # Powers
1
>>> 2**2
4
>>> 2**3
8
>>> 2**4
16
```

```
>>> pow(2,2)   # Another way for calculating powers
4
>>> pow(2,3)
8
>>> pow(2,4)
16
>>> pow(3,2)
9
>>> pow(3,3)
27
```

Divisions

We can also do divisions. Here things may *look* different depending on whether your python interpreter is version 2. something or 3. something.

- **Python 2.x**:

  ```
  >>> 13/2   # Integer Divisions
  6
  >>> 13/2.  # Float (Decimal) Divisions
  6.5
  >>> 13./2
  6.5
  >>> 13./2.
  6.5
  ```

  The first example works as follows: python interprets the 13 and the 2 as integers. Then its division just shows the integer value, i.e., it ignores the remainder and thus the decimals.

  However, if any of the two numbers has a "dot", e.g., `13.`, then python shows the full division, that is, including also the decimals.

- **Python 3.x**: All previous examples yield `6.5`. If we just want the integer results we need to write `python int(13/2)`

  ```
  >>> int(13/2)
  6
  ```

Modulo (remainder)

Remainder (or modulo operation): We saw last year what the mod operations means, namely, the remainder of dividing two integers. Let's see it again.

Say we want to distribute 8 apples among 5 students, with the constraint that we aren't allowed to cut pieces of an apple, but give them away as a whole. What's then the number of apples each student gets? Clearly, it's only 1 **and there will be 3 apples *remaining***!

We can state this by saying that *the remainder of dividing 8 by 5 is 3*, and in mathematics we write this as `8 % 5 = 3`.

Python can do this as well:

```
>>> 8%5
3
```

Assignments

In JavaScript we saw how to define variables and assign them a value. For instance, `var x = 3`. Here `x` is a label that refers to the value 3 stored somewhere in the memory of the computer.

Numbers

Here is the way to do that in python

```
>>> x = 3
>>> x
3
```

Thus we can define several variables and do operations with them:

```
>>> x =3
>>> y = 7
>>> x+y
10
>>> x*y
21
>>> y/x
2           ( in Python 2. something)
2.33333     ( in Python 3. something)
>>> float(y)/x
2.33333     ( in all versions of Python)
>>> y**x
343
>>> x**y
2187
```

We see that using `float(y)` is like adding the dot after 7. That is, `float(y)/x` is like doing `7./3` and this, we have seen above, gives the result with decimals.

Strings

We can also assign to variables a string of characters

```
>>> msg = "Hello World!"
>>> msg
'Hello World!'
```

There is also a way to "add" strings. It's not really called "adding strings" but ***concatenating*** strings. Let's see an example

```
>>> msg1 = "hi"
>>> msg2 = "jo"
>>> msg1 + msg2
'hijo'
>>> msg2 + msg1
'johi'
```

As we can see, concatenating means we append the second string at the end of the first one.

For Loops

In JavaScript we saw how to write a for-loop:

```
var sum = 0 ;

for(var i = 1 ; i<11 ; i++){
   sum = sum + i
}

//this will show the final amount of adding the integers from 1 till 10 which is 55.
alert(sum)
```

In Python we also have for-loops, although the syntax is slightly different.

In order to see for-loops in Python we need to talk first (or at the same time) of the command (*function*) `range`.

The `range` command

Python has multiple built-in "commands", or as they are technically called, *functions* (we'll see later what these are).

A very useful one is `range`.

`range(0,3)` provides a *list* of integers starting on the value 0 and ending on 2 -one less that the second number!

The following examples will easily clarify what it does.

Printing the first ten integers

The following is an example of a `for` loop that *iterates* over a list of integers, in this case, from 1 till 10:

```
>>> for i in range(1,11):
... print(i)
...
```

```
1
2
3
4
5
6
7
8
9
10
>>>
```

Important Remarks:

1. The line containing the `for` keyword ends in a colon `':'`
2. The next line **is indented by pressing the `tab` key once** This is very important: *without the indentation this code will not work*!!

   ```
   >>> for i in range(1,11):
   ... print(i)
   File "<stdin>", line 2
   print(i)
       ^
   IndentationError: expected an indented block
   >>>
   ```

3. `range(1,11)` provides a list of integers **starting at 1 and ending at 10** (and not 11) !!

   That is, it goes from 1 to 11, but excluding this last value.

Question: What will this code do?

```
>>> for i in range(10,3):
...   print(i)
...
```

Answer: Nothing. The final value provided to range is smaller than the initial value. Whence, there is **no** range of increasing integers from 10 to 3; that would be a decreasing sequence instead!

Question: What will this code do?

```
>>> for i in range(3,10):
...   print(i)
...
```

Answer: Prints all integers from 3 to 10 without including 10: 3, 4, 5, 6, 7, 8 and 9.

Summing the first 10 integers

```
>>> sum = 0
>>> for i in range(1,11):
...     sum = sum + i
...
>>> sum
55
>>>
```

Summing the first ten powers of 2

We want to calculate the following sum: $2^1+2^2+2^3+2^4+2^5+2^6+2^7+2^8+2^9+2^{10}$.

This is a way to do it in Python:

```
>>> sum = 0
>>> for i in range(1,11):
...     sum = sum + 2**i
...
>>> sum
2046
>>>
```

Writing and Running Programs

The Python interpreter comes in very handy for testing short code and ideas, but it's totally impractical to write any program with more than a few lines!

But even small, but useful programs will require a few lines at least, as we have seen. Furthermore, we would like to save time and have not have to repeat writing the same code everytime we need to do the same or similar calculations. Clearly, we cannot use the Python interpreter for this.

Instead, *we write our code using a text editor and save it into a file with extension* `.py`. Let's say we saved it into a file called `myprogram1.py` which we put inside a folder called `Python` which in turn lives inside our `CSLab` folder.

Then, we open the terminal, change directory (command called `cd`, remember?) into that folder `Python` and from there we type

```
python myprogram1.py
```

and press enter.

In order to cover the cases of both Windows and Mac/Linux, the way to type that command is **pressing the `tab` key after `python`**! (right after the letter 'n').

Conditional Expressions

Conditional expressions allow us to make choices. In order to so we need *a special* ***type*** *of value* called a **boolean type**. In most programming languages, this type is denoted as simply `bool`.

If a variable, say `itRains`, is a of type `bool`, then it can only have two possible values: `True` or `False`. In python, the capital T and F are required!

We already saw Boolean gates (or Boolean operators). We saw things like `and, or, not, xor,...`. Python includes some of such operators which we can use to build elaborate conditions.

Examples:

1.
```
>>> not True
False
>>> not False
True
>>> x = not False
>>> print(x)
True
```

2.
```
>>> usuallyGoodStudent = True
>>> cheated = True
>>> mark = 95
>>> if usuallyGoodStudent and not cheated:
...     mark = 100
...
>>> if usuallyGoodStudent and cheated:
...     mark = 10
...
>>> print(mark)
```

Puzzle: What will be printed?

We have also relational and equality operators for comparing values. These are given in the following table

Symbol	Operation
>	Greater than
<	Smaller than
>=	Greater than or equal to
<=	Smaller than or equal to
==	Equal to
!=	Not equal to

In JavaScript we saw than an `if-then-else_if-else`` conditional`

`statement can have that`else`and`else if‘ part that is executed only if the previous conditions is false.

In Python we also have those two constructions with the only difference that we don't write `else if` but `elif`.

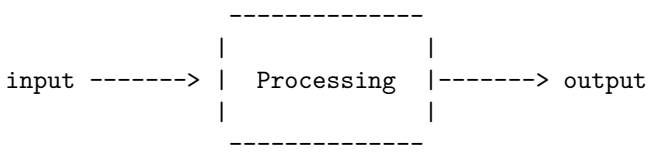
Examples:

1.
```
if mark == 100:
    print("Awesome!")
elif mark >= 90 and mark < 100:
    print("Excellent!")
elif mark >= 75 and mark < 90:
    print("Very Good!")
elif mark >= 50:
    print("Good!")
elif mark >= 50:
    print("Good!")
else :
    print("You can do it. Let's try again!")
```

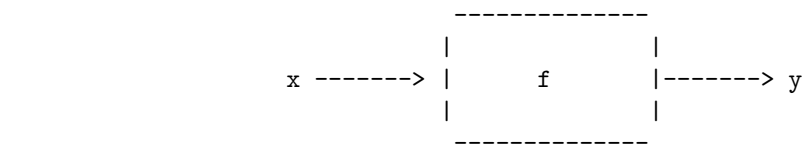
Functions

What is a *function*?

We already saw what the gist of a computer is. It's the so-called *von Neumann architecture* which at its simplest form can be summarized by the following picture



Amazingly enough, a function f can be described by the same picture



Even more amazing is the fact that functions are **everywhere around us**!

A function is like a manufacturing plant: it takes some raw material in, it process it and finally delivers a product!

Examples:

1. A petroleum oil refinery takes in oil and produces gasoline.
2. A sugar refinery takes in sugar canes and produces crystallized sugar, that is, the sugar we used at home.
3. A car manufacturing plant takes in several materials as well as already built pieces and produces a car.

Examples of functions

Function in Mathematics

When we write in math $y = 2\,x$, x denotes a *variable* that can take on different values and for each of those we get a number for y. For instance, when $x = 3$, y becomes 6. Other examples: $y = 3\,x^2 - 17\,x + 4$, $y = 2^x$,...

Clearly, in all these cases, the value of y **depends** on that of x. There is a shorter way to express this last sentence: Mathematicians say that y **is a function of** x.

Mathematicians, who are known to be extremely lazy creatures, do not even *write* such a sentence; it's too long for them. Instead of writing "y is a function of x" they write it as

$$y = f(x)$$

Somehow we can read this literally as "*y is equal to a function, called 'f', of x*". But, again, a mathematician loves to write a sentence of only 6 characters instead $y = f(x)$...

There are many different functions, and we could give each of them a different **name**. In the previous case, that name was quite a short one (remember, very lazy these mathematicians!), namely, 'f'!

Examples:

1. $y = f(x) = 2\,x$, $y = g(x) = 3\,x^2 - 17\,x + 4$ or $y = h(x) = 2^x$, where the names of the functions are f, g and h, respectively.
2. $y = area\_of\_rectangle(a, b) = a * b$, where the (for a mathematician, very painful) name of the function would be "area_of_rectangle" and the *variables* a, b would denote the lengths of each side of a rectangle. The mathematician way of writing this would, however, be much shorter: $y = g(a, b) = a * b$!

Functions in Python

We have already seen a few examples. These come *for free* with Python - technically we say they are (already) *built-in*:

1. `print( "Hello world")`: This prints the string `'Hello world'` on screen. If we write `print(3)` it prints the number 3.

The input in these examples are the string "Hello World" and the number 3. The output in each case is *what we see on the screen.*

2. `abs(-8)` *gives* 8; `abs(8)` *gives* 8; thus `abs(x)` *gives* the *absolute value* of a number, that is, it ignores any possible sign. The input is here the number x and the output is its value without sign.

3. `min(3,6)` *gives* 3; `min(3,-1,7,0,0.1)` *gives* -1; whence `min(x,y,z)` gives the smallest (most negative) of x, y and z. Whence, `print( min(3,5) )` prints 3. The input in the first example is two numbers, 3 and 6 and the output is 3; in the second example, the input is five numbers, 3,-1,7,0 and 0.1 and the output is -1.

4. Analogously, `max(3,-1,7,0,0.1)` *gives* the largest (most positive) of all provided numbers, which in this case is 7.

5. `round(3.141516,2)` *gives* the decimal number 3.141516 rounded off to only 2 decimal digits, that is, it *gives* 3.14; `round(3.141516,3)` *gives* then 3.142, because the 1 after the 4 gets rounded up to 2 due to the following 5; `round(3.141516,0)` *gives* 3 and `round(3.141516,1)` *gives* 3.1 as the 4 does not round up the first 1.

Input and output

In all previous math examples, we say that x (or a and b) is the **input** to the function, and y is the **output** delivered by that function.

In all the previous python examples, `print` `abs`, `min`, `max` and `round` are *functions* that take one or more inputs and deliver one output.

Examples:

1. If $y = f(x) = 2\,x$, when we calculate $f(3)$, the input is 3 and the output is 6.
2. If we write in python `y=min(3,x)` for every input value x, the output y will be either 3 (if x is equal or larger), or else x (if x is smaller than 3).

Building our own functions in Python

Let's see this by example.

A first simple function

Let's write the function given by $y = f(x) = 2\,x$ in python. The way to do this is as follows.

First we need to **define** the function. This means telling the computer what the function should do with its input.

```
def f(x):
    return 2*x
```

The way to type it is:

1. Write the first line `def f(x):`.
2. After the colon, press enter and **then press the tab key** in order to **indent** all the lines that define the function!
3. After the last line of the definition, press enter twice.

Here `def` and `return` are special *keywords*. The first, `def` lets the computer know that what follows is the definition of a function; the second, `return`, establishes *what the output will be*.

In order to put this function in use we simply write for instance

```
y=f(3)
print(y)
```

and the value of y that gets printed will be 6.

A function's code

Another example:

```
def diff2(x,y):
    d = abs(x-y)
    return f(d)

y = diff2(7,12)
print(y)
```

The value that gets printed is 10.

This is how it works: in the first line we calculate the absolute value of the difference between 7 and 12. That's 5. In the last line, we first make a function call to `f` with the value of that difference, i.e., with 5. This function returns the double of its input, whence it will return a 10. Finally, `diff2` returns this 10, which gets assigned to the variable `y` and is then printed to screen.

Things to notice:

1. **Name of the function**: In this case the name is "`diff2`"
2. **Number of *parameters***: This functions is defined with 2 input parameters denoted by the labels x and y.
3. **Indentation of lines after the colon**!! This is mandatory, as it is the only way for the computer to know where the definition of the function ends!!
4. **A function's code**: A function's code comprises all the lines after the `def` line and up to, and including, the last indented line. Whence, in this case the code of the function `diff2` is:

   ```
   d = abs(x-y)
   return f(d)
   ```

5. **Function call**: A function call is when we use a function with specific values, like 7 and 12 in the line

   ```
   y=diff2(7,12)
   ```

   These specific values are called ***arguments*** *of the function call*, or, in short, arguments of the function, and even shorter, *arguments*! Here the arguments are 7 and 12.

6. **Function calls *inside* functions!**: We can use any function previously defined in any part of our code, *even inside other functions*!! In this example, *we make 2 function calls inside* `diff2`: one to the built-in function `abs` and the other to our function `f` defined above. Here in lies the power and beauty of using functions in our code!!

This example hints at the tremendous advantage of using functions: Defining functions is like building a toolbox. Any subsequent problem can be solved by *using* one or several of those tools, *without having to build them again each time*!

Solved problems

1. Write a function called `sum_n` that takes one argument, an integer `n` and returns the sum of all numbers from 1 till `n`, both included. What's the output of `sum_n(23)`?

   Solution:

   ```
   def sum_n(n):
       sum = 0
       for i in range(1,n+1):
               sum = sum + i
       return sum
   y = sum_n(23)
   print(y)                    # y = 276
   ```

2. Write a function called `fact` that takes as input one integer `n` and returns the *factorial of* n, $n!$. Note: The factorial of an integer `n`, denoted as $n!$, is the product of all integers from 1 till n both included. Example: $2! = 2 \cdot 1 = 1$, $3! = 3 \cdot 2 \cdot 1 = 6$, or $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. By definition, the factorial of zero is 1, i.e., $0! = 1$.

   Solution:

   ```
   def fact(n):
   if n == 0 : return 1
       p = 1
       for i in range(1,n+1):
               p = p * i
       return p
   ```

```
y = fact(5)
print(y)                        # y = 120
```

3. Write a function called `fibn` that takes as input one integer `n` and returns the n-th number of the Fibonacci sequence. Note: The Fibonacci sequence starts as follows: 1 1 2 3 5 8 13 21 34 55 89 144 233 . . .

   Solution:

```
def fibn(n):
    fb1 = 1
    fb2 = 1
    if n < 3 : return 1
    for i in range(1, n-1 ):
            t = fb2
            fb2 = fb1 + fb2
            fb = t
    return fb2

y = fibn(13)
print(y)                        # y = 233
```

   Advanced: There is an alternative solution, using somewhat more of an advanced technique called *recursive programming*, and the function we code this way is an example of a *recursive function*. This entails the use of the function we are defining ***in the very same code that defines it***.

   This can sound *paradoxical*, like when we use a concept in the definition of that same concept, or like Esher's picture of a hand drawing a second hand which draws the first hand, which draws the second hand, which draws. . .

   How does it work? The key is to include first a *base condition* that breaks that infinite loop **without using recursion**, i.e., without calling itself.

```
def fibn(n):
    if n < 3: return 1   # This is the base case. No call to `fibn` here.
    return fibn(n-1) + fibn(n-2)
```

   You may understand how this can work by trying to recite out loud the definition of the factorial of, say, 5 in a recursive way. This would be as follows: "5 factorial is 5 times 4 factorial; 4 factorial is 4 times 3 factorial; 3 factorial is 3 times 2 factorial; 2 factorial is 2 times 1 factorial; 1 factorial is 1 times 0 factorial". But, wait!, here the *apparent* infinite loop breaks! The factorial of 0 is simply 1!!

   Now we can trace back the full result. Read those sentences for the last till the first: 1 factorial is 1 times. . . 1, whence 1; 2 factorial is 2 times 1 factorial, whence 2 times 1, thus 2; 3 factorial is 3 times 2 factorial, whence

3 times 2, thus 6; 4 factorial is 4 times 3 factorial, whence 4 times 6, thus 24; and finally, 5 factorial is 5 times 24, thus 120.

4. Write a function called `fibsum` that takes as input one integer `n` and returns the sum of the first n-th Fibonacci numbers

   Solution 1: We will use the function `fibn` that we already wrote.

```
def fibsum(n):
    sum = 0
    for i in range(1,n+1):
            sum += fib(i)
    return sum

print( fibsum(4) )          # prints 7
print( fibsum(13) )         # prints 609
```

   Solution 2: We will write the solution from scratch, i.e., without using the function `fibn` we already defined.

```
def fibsum(n):
    fb1 = 1
    fb2 = 1
    sum = 2
    if n == 1 : sum = 1
    for i in range(3, n+1):
            sum = sum + fb1 + fb2
            t = fb2
            fb2 = fb1 + fb2
            fb1 = t
    return sum
```

Clearly, the first solution is much easier to follow and thus understand than the second one. This illustrates the importance and usefulness of writing functions and using them instead of repeating their code over and over each time we need to do the same task!