Elementary Python for OT-2 API

May 2, 2021

Table of Contents

1 Introduction

In this workshop we will cover the basic elements of programming in Python. Only materials that are absolutely essential to understanding and programming the OT-2 API will be covered. This workshop is intended for people with little to no programming experience.

2 Objectives

By the end of this workshop, we hope participants are able to:

1. Set up an IDE for coding in Python, and put to use the features that faciliate coding and debugging.
2. Understand the common data types in Python and when to use which.
3. Perform simple calculations and modify strings.
4. Understand the concept of function and where to check their arguments.
5. Write for loops to automatically repeat blocks of codes.
6. Understand superficially the concept of Object-Oriented Programming.
7. Make use of knowledge of points 2-5 above to solve practical issues in using the OT-2.

3 Disclaimer

I have limited experience in programming and received little formal training. Coding practices may be unorthodox and even heretic in the eyes of properly trained programmers.

The information that we will go over is filtered specifically for understanding the OT-2 API. Many other useful functions are omitted simply because they are non-essential from the API's perspective. Those who wish to make the full use of Python + the robot would benefit greatly by attending

Python courses offered by the Software Capentry in the University of Edinburgh or by the Edinburgh Genomics.

Please forgive my ocassional loss of professionalism in language.

4 System Setup

4.1 Installing Spyder

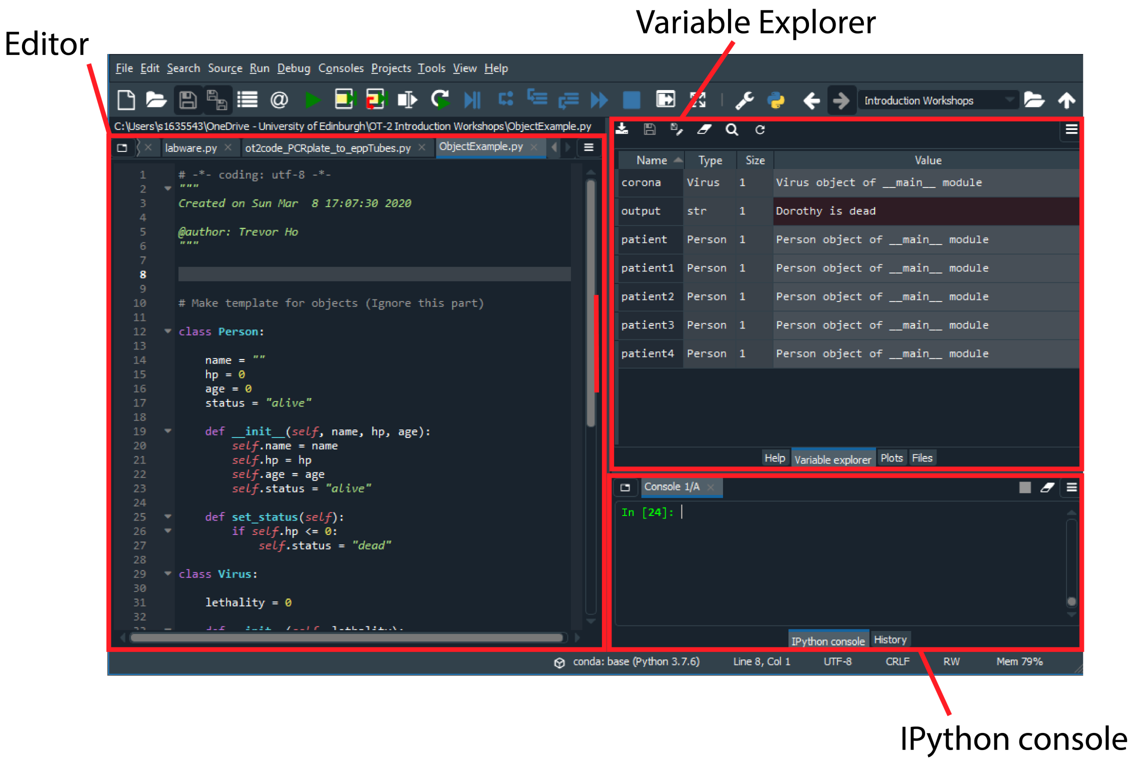
It is best to make use of an Integrated Development Environment (IDE) when writing codes. My personal experience suggests that Spyder is most friendly to Python beginners and its Variable Explorer is tremendously helpful, both in debugging and in understanding how the code works.

The relatively fool-proof way to get Spyder is to download and install Anaconda.

Note: I strongly recommend you NOT to install the app Kite. It does more harm than good in my experience

From time to time you might need to upgrade your IDE or your packages. To do so, type `conda update --all` inside the Spyder IPython console, or follow the instructions from Anaconda.

5 Navigating within Spyder



```
C:\Users\s1635543\OneDrive - University of Edinburgh\OT-2 Introduction Workshops\ObjectExample.py
```

```python
# -*- coding: utf-8 -*-
"""
Created on Sun Mar  8 17:07:30 2020

@author: Trevor Ho
"""



# Make template for objects (Ignore this part)

class Person:

    name = ""
    hp = 0
    age = 0
    status = "alive"

    def __init__(self, name, hp, age):
        self.name = name
        self.hp = hp
        self.age = age
        self.status = "alive"

    def set_status(self):
        if self.hp <= 0:
            self.status = "dead"

class Virus:

    lethality = 0

```

Name	Type	Size	Value
corona	Virus	1	Virus object of __main__ module
output	str	1	Dorothy is dead
patient	Person	1	Person object of __main__ module
patient1	Person	1	Person object of __main__ module
patient2	Person	1	Person object of __main__ module
patient3	Person	1	Person object of __main__ module
patient4	Person	1	Person object of __main__ module

```
In [24]:
```

5.1 Useful shortcuts in Spyder

1. F5 = Run entire script
2. Ctrl/Cmd + Return/Enter = Run cell

3. Ctrl/Cmd + 1 = Comment/De-comment

Cells are partitioned by the comment #%% in the Spyder Editor.

6 Getting started

6.1 print()

The print() function is your best friend. You get to know what's going on by asking the computer to show you what you are interested.

[1]:
```
print("Hello World")
```

```
Hello World
```

6.2 Variable assignment

The equal sign = in almost all programming languages denote an assignment of value to a variable.

[2]:
```
a = 1000
print(a)
```

```
1000
```

[3]:
```
b = "OT-2"
print(b)
```

```
OT-2
```

Double assignment allows two assignments in a single line.
You can also do triple or mulitple assignments.

[4]:
```
a, b = 1000, "OT-2"
print(a)
print(b)
```

```
1000
OT-2
```

Note that if you don't do an assignment in the line, the console will give the output directly.

[5]:
```
a
```

[5]:
```
1000
```

[6]:
```
b
```

[6]:
```
'OT-2'
```

6.3 Comments

Comments are your explanations of what the code does. It helps other people to understand what you are trying to do, and helps you remember why you code in a particular way. The following example is just an illustration. Comments for self-explanatory codes are redundant.

[7]:
```
# This code calculate Fluorescence / OD
fluo = 5000
od = 2.0
fluo_OD = fluo / od
print(fluo_OD)
```

```
2500.0
```

[8]:
```
# # !"£$%^&*()_+"      * Note that once you have a hashtag, whatever that comes␣
 ↪afterwards are all comments.
# You have to start a new line without a hashtag to resume coding
fluo
```

[8]:
```
5000
```

One useful thing to do with comments is to "comment-out" a section of the code, that is, not run that part only.

[9]:
```
a = 1000
b = 200
a = a/b
print("Without using comment-out:")
print(a)
```

```
Without using comment-out:
5.0
```

[10]:
```
a = 1000
b = 200
# a = a/b
print("Using comment-out:")
print(a)
```

```
Using comment-out:
1000
```

7 Commonly used variable types (officially: data types)

7.1 Some basic data types

7.1.1 integer (int)

[11]:
```python
var_int = 4000
print(var_int)
print(type(var_int))
```

```
4000
<class 'int'>
```

7.1.2 strings (str)

for texts and characters

[12]:
```python
var_str = "OT-2"
print(var_str)
print(type(var_str))
```

```
OT-2
<class 'str'>
```

7.1.3 floating point numbers (float)

In an over-simplified sense, any other number that carries decimal points

[13]:
```python
var_float = 10123598.8562545
print(var_float)
print(type(var_float))
```

```
10123598.8562545
<class 'float'>
```

7.1.4 booleans (bool)

Only takes True or False, equivalent to 1 or 0.

[14]:
```python
var_bool1 = True
var_bool2 = False
print(var_bool1)
print(var_bool2)
print(type(var_bool1))
```

```
True
False
<class 'bool'>
```

It is also possible to infer the type of the variable in the console by calling it directly

```
[15]: var_int
```

```
[15]: 4000
```

```
[16]: var_str
```

```
[16]: 'OT-2'
```

```
[17]: var_float
```

```
[17]: 10123598.8562545
```

```
[18]: var_bool1
```

```
[18]: True
```

7.2 Type conversion

In some cases it is useful to convert string into numbers, and vice versa

```
[19]: str(89006542)
```

```
[19]: '89006542'
```

```
[20]: int('85678')
```

```
[20]: 85678
```

If you string contains alphabets, conversion to an int or float will fail

```
[21]: int('A1')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-21-d5e71966c3c3> in <module>
----> 1 int('A1')

ValueError: invalid literal for int() with base 10: 'A1'
```

Conversion of a float in string format to int is also prohibited

```
[22]: float('85.67')
```

```
[22]: 85.67
```

```
[23]: int('85.67')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-23-515ef414cde9> in <module>
----> 1 int('85.67')

ValueError: invalid literal for int() with base 10: '85.67'
```

Conversion of float to int will remove the digits behind the decimal point

[24]:
```
int(1000.786)
```

[24]:
```
1000
```

8 Lists and dictionaries

If you do a PCR and only have one sample you use a single PCR tube. If you deal with 7-8 samples that are ordered you use PCR tube strips. If you have hundreds of samples you use 96-well or 384-well plates. The key point is that your sample locations are **indexed**. You do the same in programming.

8.1 list

A list is equivalent to PCR tube strips, just that they have (nearly) infinite length. You label one strip with one name.

Items are delimited by a comma.

[25]:
```
var_list = [1, 2, 3, 4, 5, 6]
var_list
```

[25]:
```
[1, 2, 3, 4, 5, 6]
```

To create an empty list, just leave it empty but with brackets

[26]:
```
var_list = []
var_list
```

[26]:
```
[]
```

You can literally put anything inside a list

[27]:
```
a = 10
var_list = [1, a, 100, 'text', True] # A mix of int, var, str, and bool
var_list
```

[27]:
```
[1, 10, 100, 'text', True]
```

You can put a list inside a list

```
[28]: list1 = [5, 6, 7]
      list2 = [1, 2 , 3, list1]
      list2
```

```
[28]: [1, 2, 3, [5, 6, 7]]
```

If you have a long list, you can arrange them vertically in the code.
Ending a list with a comma is fine.

```
[29]: int_list = [
          1005719631856,
          5746381643542,
          8434216572496,
          2491294949454,
          21975214621121,
          1964943136274,
          1984249121000,
          149219542792464,
      ]
      int_list
```

```
[29]: [1005719631856,
       5746381643542,
       8434216572496,
       2491294949454,
       21975214621121,
       1964943136274,
       1984249121000,
       149219542792464]
```

All items inside a list has an index for retrieving the item. A list is ordered - the item order is preserved when the list is called. Using the example of `var_list`:

index	item
0	1
1	10
2	100
3	‘text’
4	True

This allows you to get retrieve specific items from the list using the `[]` notation.

Note that the first item of a list has the index of 0, not 1. Think of a list as a building and index 0 is the “Ground Floor”.

```
[30]: var_list[0]
```

```
[30]: 1
```

```
[31]: var_list[3]
```

```
[31]: 'text'
```

To replace an item in the list, reassign it.
Let's replace the 'text' item with index=3 to '1000'

```
[32]: var_list[3] = 1000
      var_list
```

```
[32]: [1, 10, 100, 1000, True]
```

Trying to call the an item beyond indicies that correspond to an item will lead to an `IndexError`. If you see this error, you must be counting the list wrong.

```
[33]: var_list[5]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-33-12cebd302046> in <module>
----> 1 var_list[5]

IndexError: list index out of range
```

Counting backwards? 2nd item from the end of the list? Put a minus symbol - before your index 1

```
[34]: var_list[-2]
```

```
[34]: 1000
```

Slice a list by using the "slice operator" :

```
[35]: # Get items from index = 2 till the end of sequence
      var_list[2:]
```

```
[35]: [100, 1000, True]
```

```
[36]: # Get items from the beginning till item of index = 2
      # In other words, var_list[:3] is equivalent to var_list[0:3]
      var_list[:3]
```

```
[36]: [1, 10, 100]
```

```
[37]: var_list[3:5]
```

[37]:
```
[1000, True]
```

Take note! [3:5] actually gives items from index=0 to index=2. So it is actually getting item_of_start_index to item_of_(end_index - 1) !

Combining a negative index with the splice operator allows you to get multiple items from the end of the list

[38]:
```
var_list[-3:]
```

[38]:
```
[100, 1000, True]
```

***Beware that when couting backwards, the last item has an index of -1, not -0.**

To get the length of the list, use the `len()` function, with the list as the input argument

[39]:
```
len(var_list)
```

[39]:
```
5
```

To add items to a list (aka. grow a list), use the `list.append()` function

[40]:
```
item_to_add = 'new'
var_list.append(item_to_add)
var_list
```

[40]:
```
[1, 10, 100, 1000, True, 'new']
```

[41]:
```
# The list should now have a length of 6 instead of 5
len(var_list)
```

[41]:
```
6
```

To concatenate two or more lists, simply use the `+` operator

[42]:
```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = [7, 8, 9]
long_list = list1 + list2 + list3
long_list
```

[42]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

8.2 dictionary (dict)

A dictionary can be understood as a list with unique “name tags” to every item it stores. While a list is denoted by square brackets, a dictionary is denoted by curly brackets.

[43]:
```
chores = {'Monday': "Clean gel tank",
            'Tuesday': "Autoclave tips",
            'Wednesday': "Clear bins"
           }
chores
```

```
[43]: {'Monday': 'Clean gel tank',
       'Tuesday': 'Autoclave tips',
       'Wednesday': 'Clear bins'}
```

Items are retreived in a similar way as lists, just that there are no indicies but only “name-tags”, which are called **keys**.

```
[44]: chores['Monday']
```

```
[44]: 'Clean gel tank'
```

If you try to call a key that is absent from the dictionary, you will get an error

```
[45]: chores['Sunday']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-45-50c1074f10b3> in <module>
----> 1 chores['Sunday']

KeyError: 'Sunday'
```

To avoid running into error, use the `dict.get(key)` function

```
[46]: print(chores.get('Monday'))
```

```
Clean gel tank
```

```
[47]: print(chores.get('Sunday'))
```

```
None
```

Now, the reason why it is called a dictionary, is that like a dictionary, you can’t have two entries of the same word.
If you insist on doing so, the most recent assignment will overwrite the previously assigned key.

```
[48]: # Note how the first definition becomes "erased" in the output

var_dict = {
    'table': "a piece of furniture with a flat top and one or more legs (Google␣
 ↪def 1)",
    'table': "a set of facts or figures systematically displayed, especially in␣
 ↪columns (Google def 2)"
}
```

```
var_dict
```

```
[48]: {'table': 'a set of facts or figures systematically displayed, especially in
      columns (Google def 2)'}
```

To create an empty dictionary, put nothing inside the curly brakets

```
[49]: null_dict = {}
      null_dict
```

```
[49]: {}
```

To add a `key: item` pair to a dictionary, just assign it to a new key

```
[50]: chores['Thursday'] = "Refill MilliQ water tank"
      chores
```

```
[50]: {'Monday': 'Clean gel tank',
       'Tuesday': 'Autoclave tips',
       'Wednesday': 'Clear bins',
       'Thursday': 'Refill MilliQ water tank'}
```

Be careful though, if a key alreayd exist, trying to add a new `key: item` pair with the same key will overwrite the previous item.

To concatenate two dictionaries, use the `dict.update()` function. The input argument must be a dictionary.

```
[51]: new_chores = {
          'Friday': "Buy beer for happy hour",
          'Saturday': "Hang over",
                   }
      chores.update(new_chores)
      chores
```

```
[51]: {'Monday': 'Clean gel tank',
       'Tuesday': 'Autoclave tips',
       'Wednesday': 'Clear bins',
       'Thursday': 'Refill MilliQ water tank',
       'Friday': 'Buy beer for happy hour',
       'Saturday': 'Hang over'}
```

The same function can also be used to add a single key:item pair, making it a very versatile function. I prefer using it over the assignment method.

```
[52]: chores.update({'Sunday': "Do nothing"})
      chores
```

```
[52]: {'Monday': 'Clean gel tank',
       'Tuesday': 'Autoclave tips',
       'Wednesday': 'Clear bins',
       'Thursday': 'Refill MilliQ water tank',
       'Friday': 'Buy beer for happy hour',
       'Saturday': 'Hang over',
       'Sunday': 'Do nothing'}
```

9 Notes on doing basic maths

Simple addition, subtraction, multiplication is just like that of Excel

```
[53]: 5 + 10 * 20
```

```
[53]: 205
```

The above example takes only int as input and you get back an int as output. Put a float anywhere inside and you get back a float.

```
[54]: 5.0 + 10 * 20
```

```
[54]: 205.0
```

Whenever you do a division, you always get back a float, even if your inputs are all int and your dividend is divisible by your divisor:

```
[55]: 100 / 10
```

```
[55]: 10.0
```

The proper way to do power is not through the ^ sign, but rather, `**`

```
[56]: 2^4
```

```
[56]: 6
```

```
[57]: 2 ** 4
```

```
[57]: 16
```

10 Basic string manipulation

String can be seen as list objects consisting of only characters, numbers and symbols. So, you can use some of the same functions from list on strings.

10.1 Empty string

```
[58]: null_str = ""
```

10.2 Concatenation

```
[59]: hello_world  = "Hello" + " " + "World"
      hello_world
```

```
[59]: 'Hello World'
```

10.3 Substring using slice

```
[60]: hello_world[:5]
```

```
[60]: 'Hello'
```

```
[61]: hello_world[-3:-1]
```

```
[61]: 'rl'
```

10.4 ~~.append()~~

Note that there is no such function `.append()` for str

```
[62]: hello_world.append(" Goodbye World")
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-62-b9e99d8024a8> in <module>
----> 1 hello_world.append(" Goodbye World")

AttributeError: 'str' object has no attribute 'append'
```

10.5 .split()

The `.split()` function takes a user-determined delimiter, removes the delimiter from the str, and return the rest of the parts as a list.

```
[63]: hello_world.split(" ")
```

```
[63]: ['Hello', 'World']
```

```
[64]: "University of Edinburgh".split(" ")
```

```
[64]: ['University', 'of', 'Edinburgh']
```

The delimiter can be anything, so long as it is inside the str

```
[65]: split_str = "University of Edinburgh".split(" of ")
      split_str
```

```
[65]: ['University', 'Edinburgh']
```

The individual words/strings/stretches of characters can then be retrieved through the list's indicies

```
[66]: city_name = split_str[1]
      city_name
```

```
[66]: 'Edinburgh'
```

A combinatoin of double assignment and `.split()` works magic in Python.

```
[67]: my_name = "Trevor Ho"
      first_name, last_name = my_name.split(" ")
      first_name
```

```
[67]: 'Trevor'
```

```
[68]: last_name
```

```
[68]: 'Ho'
```

11 Simple custom functions

The function in computer science has its roots in mathematical functions like y = f(x), y is output, f is the function name, x is the input. It is very similar in Python.

Example:

```
[69]: def divide(dividend, divisor):
          quotient = dividend / divisor
          return quotient

      divide(1000,10)
```

```
[69]: 100.0
```

function name: the name that is followed by parentheses and right after the reserved word `def`
input: variables that are inside the parentheses, their proper name is **"arguments"**
output: the varaible that follows after the reserved word `return`

the output (and hence `return`) is entirely optional if you just want the function to carry out something

Important: notice that the lines below `def` are right indented. This is to tell Python that those lines belong to the function

[70]:
```
def divide(dividend, divisor):
quotient = dividend / divisor
return quotient
```

```
  File "<ipython-input-70-892f0500f8b6>", line 2
    quotient = dividend / divisor
           ^
IndentationError: expected an indented block
```

An extremely important point is that **the order of arguments matters**:

[71]:
```
divide(10,1000)
```

[71]:
```
0.01
```

However, it is ok change the order if your specify what variables are meant for what arguments

[72]:
```
divide(divisor=10, dividend=1000)
```

[72]:
```
100.0
```

Note: always write and debug your function without `def` and `return` first before encapsulating it with def and return.

12 For loop

For loop is used when you need to repeat blocks of code for a set number of times. The for loop is arguably the most important concept to master if you wish to make the most out of the robot.

Let's start with an example

[73]:
```
for i in [1,2,3]:
    print("I should do my experiments")
```

```
I should do my experiments
I should do my experiments
I should do my experiments
```

The command `print()` was executed three times, because there are 3 items within the list of `[1,2,3]`
Notice that it was indented and so was within the loop.

Simply iterating a fixed action isn't very helpful, but the for loop in Python is inherently built to be mutable at every iteration:

[74]:
```
for well in ["A1", "B1", "C1"]:
    action = "Pipette water into well " + well
    print(action)
```

```
Pipette water into well A1
Pipette water into well B1
Pipette water into well C1
```

12.1 `range()`

If you need to iterate your action 100 times, it would not make sense to manually create a list of 1 to 100. The `range(start, stop, step)` function is very useful in this situation.

[75]:
```python
# Notice that the range function creates only a range object
x = range(1,10)
x
```

[75]:
```
range(1, 10)
```

[76]:
```python
# To turn that into something useful, pass the range object into the list()␣
↪function
list(x)
```

[76]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Notice that the function does not stop at `10`, but rather, `9`. This is similar to the `:` operator in slicing a list. To obtain a list of numbers from 1 to 12, the correct input is `range(1,13)`

A combination of the `range()` function with a for loop then works magic.

[77]:
```python
for column in range(1,13): # by convention, variable names of i, j, k are very␣
↪popular in for loop creation
    action = "Pipette water into column " + str(column) + " of the destination␣
↪plate"
    print(action)
```

```
Pipette water into column 1 of the destination plate
Pipette water into column 2 of the destination plate
Pipette water into column 3 of the destination plate
Pipette water into column 4 of the destination plate
Pipette water into column 5 of the destination plate
Pipette water into column 6 of the destination plate
Pipette water into column 7 of the destination plate
Pipette water into column 8 of the destination plate
Pipette water into column 9 of the destination plate
Pipette water into column 10 of the destination plate
Pipette water into column 11 of the destination plate
Pipette water into column 12 of the destination plate
```

12.2 Nested for loops

Sometimes we have loop in more than 1 dimension, e.g. Sample 1: pipette up and down 3 times, Sample 2: pipette up and down 3 times, Sample 3: pipette up and down 3 times.

To do this, it is helpful to put one for loop inside the other.

[78]:
```
for sample in ["1", "2", "3"]:
    change_sample = "Switch to sample " + sample
    print(change_sample)
    for i in range(3):
        print("Pipette sample up and down")
```

```
Switch to sample 1
Pipette sample up and down
Pipette sample up and down
Pipette sample up and down
Switch to sample 2
Pipette sample up and down
Pipette sample up and down
Pipette sample up and down
Switch to sample 3
Pipette sample up and down
Pipette sample up and down
Pipette sample up and down
```

To further integrate the sample name and action step:

[79]:
```
for sample in ["1", "2", "3"]:
    for i in range(3):
        action = "Pipette sample " + sample + " up and down"
        print(action)
```

```
Pipette sample 1 up and down
Pipette sample 1 up and down
Pipette sample 1 up and down
Pipette sample 2 up and down
Pipette sample 2 up and down
Pipette sample 2 up and down
Pipette sample 3 up and down
Pipette sample 3 up and down
Pipette sample 3 up and down
```

13 If else statements (optional material)

13.1 Comparison operators

They are used to compare two variables. Output is a boolean.

[80]:
```
10 < 100
```

[80]:
```
True
```

[81]:
```
10 > 100
```

```
[81]: False
```

```
[82]: 15 <= 15
```

```
[82]: True
```

Comparison operator for checking equality is ==. This often get mixed up with the assignment symbol =.

```
[83]: 'Python' == 'Python'
```

```
[83]: True
```

```
[84]: 'Python' != 'Python'
```

```
[84]: False
```

Checking if item in lists and dictionaries

```
[85]: list1 = [1, 2, 3, 4, 5, 6, 7, 8]
      3 in list1
```

```
[85]: True
```

```
[86]: 9 in list1
```

```
[86]: False
```

```
[87]: chores
```

```
[87]: {'Monday': 'Clean gel tank',
       'Tuesday': 'Autoclave tips',
       'Wednesday': 'Clear bins',
       'Thursday': 'Refill MilliQ water tank',
       'Friday': 'Buy beer for happy hour',
       'Saturday': 'Hang over',
       'Sunday': 'Do nothing'}
```

```
[88]: 'Monday' in chores
```

```
[88]: True
```

```
[89]: 'March' in chores
```

```
[89]: False
```

13.2 Logical operators

[90]:
```
True and True
```

[90]:
```
True
```

[91]:
```
True and False
```

[91]:
```
False
```

[92]:
```
True or False
```

[92]:
```
True
```

[93]:
```
not True
```

[93]:
```
False
```

[94]:
```
8 >= 9 and 10 <= 11
```

[94]:
```
False
```

[95]:
```
8 >= 9 or 10 <= 11
```

[95]:
```
True
```

[96]:
```
8 >= 9 or not (10 <= 11)
```

[96]:
```
False
```

13.3 If else examples

[97]:
```
fluo1 = 16
fluo2 = 20
if fluo1 == fluo2:
    print("Same")
else:
    print("Different")
```

```
Different
```

[98]:
```
if fluo1 == fluo2:
    print("Same")
elif fluo1 > fluo2:
    print("Fluo1 larger")
else:
    print("Fluo2 larger")
```

```
Fluo2 larger
```

Indentation is important

```
[99]: if fluo1 == fluo2:
          print("Same")
          elif fluo1 > fluo2:
              print("Fluo1 larger")
          else:
              print("Fluo2 larger")
```

```
  File "<ipython-input-99-9d4b7f21bf49>", line 3
    elif fluo1 > fluo2:
       ^
SyntaxError: invalid syntax
```

14 A superficial introduction to the concept of Object-Oriented Programming (OOP)

The idea of OOP is to make virtual objects inside the computer and let them interact with each other, or ask them to do something.

A good cartoon illustration is available on a page written by FreeCodeCamp. Let's scroll until you see a cat.

The most important concept is that most virtual objects are like an object in real life: each has some properties (e.g. breed, color, gender, height, weight, etc.) and is able do something (e.g. meow, eat, sleep, etc)

Objects' properties are coded by variables within an object. In Python, they are accessed as `object.var`.
For example, `cat.breed`

Objects' abilities to do something are executed via methods. In Python, they are accessed as `object.method()`.
For example, `cat.eat()`

Take home messages: 1. If you see `sth.xyz`, without parentheses behind, `xyz` is a variable/attribute/property of the object. 2. If you see `sth.xyz()`, `xyz()` is a method that is part of the "innate" abilities of the object to do something.
3. Objects themselves can act on other objects, i.e. when an object is an argument of a method of another object.

Superficially, the terms "properties", "attributes" and "fields" are often used interchanagbly. One can also view "functions" and "methods" in a similary manner, though methods and functions are technically distinct.

The following is an example of creating objects of humans and viruses to see if the humans with some properties (`age`) can survive an infection `.infect()` by a virus:

[100]:
```python
# Make templates for objects (Ignore this part)

class Person:

    name = ""
    hp = 0
    age = 0
    status = "alive"

    def __init__(self, name, hp, age):
        self.name = name
        self.hp = hp
        self.age = age
        self.status = "alive"

    def set_status(self):
        if self.hp <= 0:
            self.status = "dead"

class Virus:

    lethality = 0

    def __init__(self, lethality):
        self.lethality = lethality

    def infect(self,person):
        person.hp = person.hp - person.age * self.lethality
        person.set_status()

# Create objects

patient1 = Person(name = "Alexander", hp = 1000, age = 50)
patient2 = Person(name = "Beth", hp = 9000, age = 25)
patient3 = Person(name = "Chris", hp = 5000, age = 17)
patient4 = Person(name = "Dorothy", hp = 7000, age = 90)

corona = Virus(lethality = 100)

# Let objects take action

corona.infect(patient1)
corona.infect(patient2)
corona.infect(patient3)
corona.infect(patient4)

# Get the outcomes
```

```python
for patient in [patient1, patient2, patient3, patient4]:
    outcome = patient.name + " is " + patient.status
    print(outcome)
```

```
Alexander is dead
Beth is alive
Chris is alive
Dorothy is dead
```

15 Final remarks (that we cannot cover today)

1. When you are stuck, Google. Chances are there is someone who had the same issue as you do and his/her question is answered on Stack Overflow.
2. There is an official guide in Python on how to style your code to improve readibility. As you code more and more frequently you will find this useful.
3. Learning how to read documentation of libraries / codes written by others will grant you the ability to learn how to code on your own.
4. Properly document your code will make them useful to yourself and others in the long run.