01_What_Is_Python

March 9, 2022

1 Class outline

Section ??
Section ??
Section ??
Section ??
Section ??
Section ??
Section ??
Section ??
Section ??

[1]:
```
from IPython.display import Image
```

2 What is programming?

- Series of instructions given to a computer to to process information
- Instructions must be written in a way the computer can understand
- Programming languages are used to write programs

So, are we done?

… not yet

According to Claude Shannon's theory information is present whenever a **signal** is transmitted between a **sender** and a **receiver**

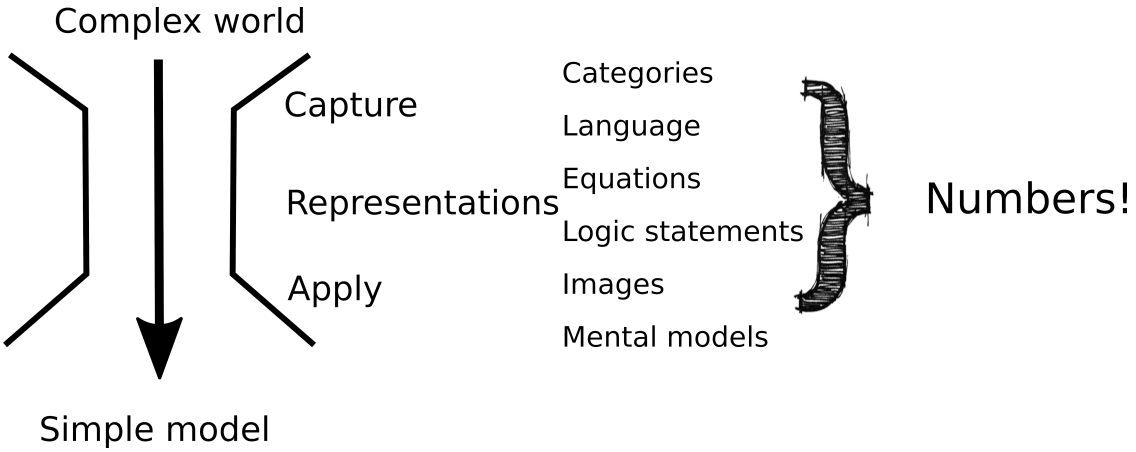
A signal can be anything from spoken words, to radio signals

2.0.1 Representations

Information is generated to facilitate human needs:

[2]:
```
Image("representation.png")
```

[2]:



2.1 Software

2.1.1 Application Software

- Word Processors
- Database s/w
- Spreadsheets
- Painting programs
- Web browsers, email programs

2.1.2 System Software

Operating Systems

- Windows
- MacOS / iOS
- Unix
- Linux
- Android

2.1.3 Programming Languages

Three major families of languages:

- Machine languages (very low level)
- Assembly languages (low level)
- High-Level languages (human like)

2.1.4 Machine Language

Comprised of 1s and 0s (remember about signals?) The ``native'' language of a computer Difficult to program -- one misplaced 1 or 0 will cause the program to fail.
Example of code:

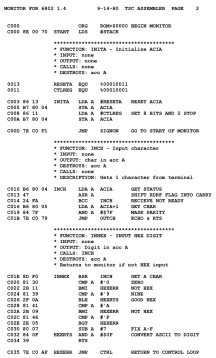
```
 1110100010101 111010101110
10111010110100 10100011110111
```

2.1.5 Assembly

Assembly languages are a step towards easier programming. Assembly languages are comprised of a set of elemental commands which are tied to a specific processor. Assembly language code needs to be translated to machine language before the computer processes it.

[3]:
```
Image(filename="assembly.png")
```

[3]:

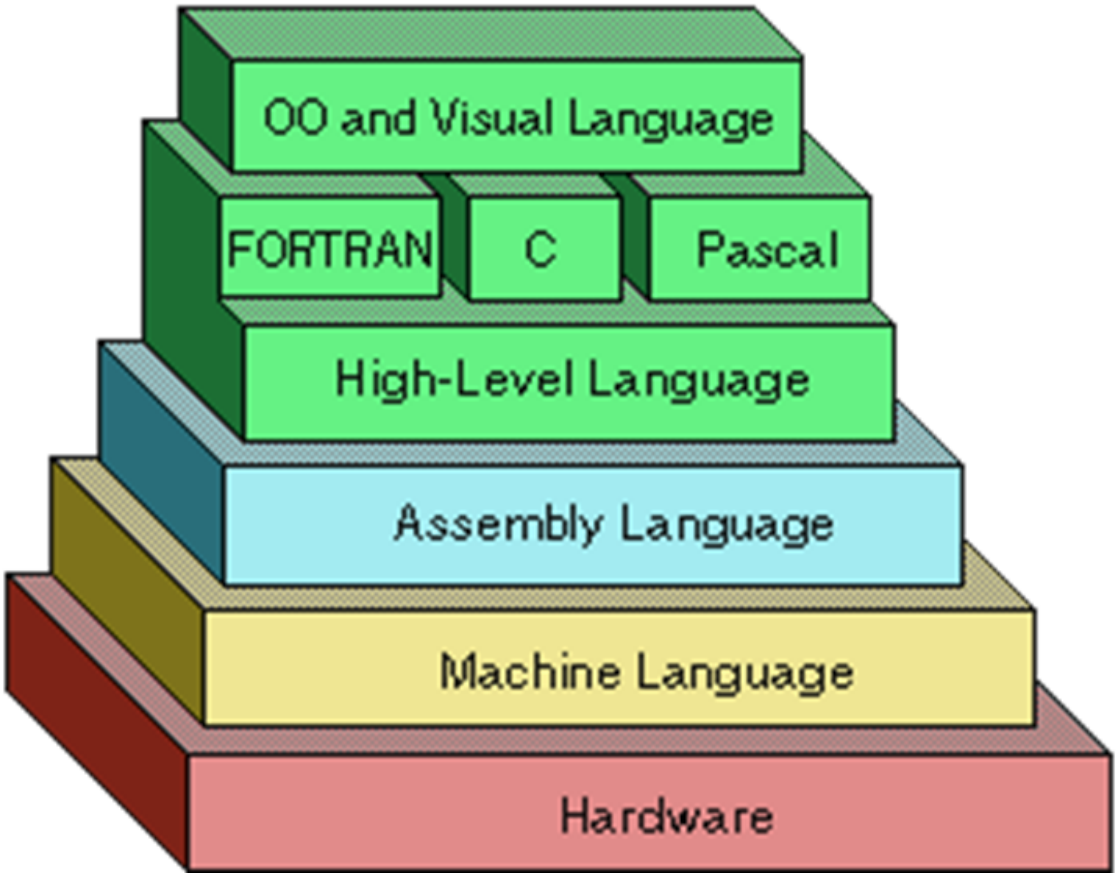


2.2 High level languages

High-level languages represent a giant leap towards easier programming.
The syntax of HL languages is similar to English.
Historically, we divide HL languages into two groups:
- Procedural languages (focus on program structure) - Object-Oriented languages (OOP)(focus on data modeling (classes))

[4]:
```
Image(filename="lan_hie.png")
```

[4]:



the pyramid is very simplified …

[5]:
```
Image(filename="languages.png")
```

[5]:

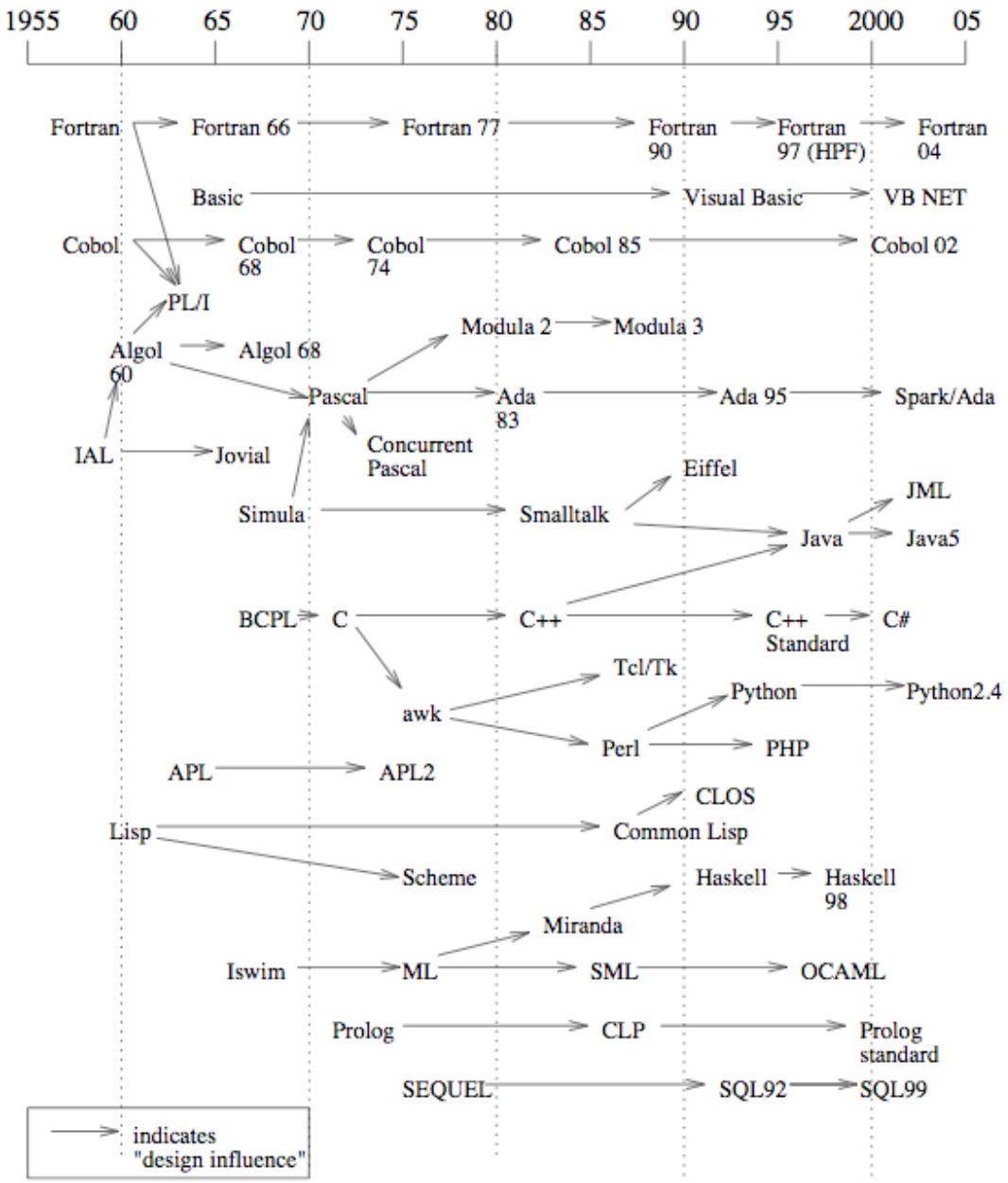


Figure 1.2: A Snapshot of Programming Language History

2.2.1 What makes a programming language

```
Programming languages have four properties:
- Syntax - Names - Types - Semantics
```

2.2.2 Syntax

The syntax of a programming language is a precise description of all its grammatically correct programs.
When studying syntax, we ask questions like:
- What is the grammar for the language? - What is the basic vocabulary? - How are syntax errors detected?

2.2.3 Names

Various kinds of entities in a program have names:
variables, types, functions, parameters, classes, objects, …

Named entities are bound in a running program to:
- Scope - Visibility - Type - Lifetime

2.2.4 Types

- A type is a collection of values and a collection of operations on those values.

Simple types: numbers, characters, booleans, …

Structured types: Strings, lists, trees, hash tables, …

A language's type system can help to:
- Determine legal operations - Detect type errors

2.2.5 Semantics

The meaning of a program is called its semantics. In studying semantics, we ask questions like:

- When a program is running, what happens to the values of the variables?
- What does each statement mean?
- What underlying model governs run-time behavior, such as function call?
- How are objects allocated to memory at run-time?

2.2.6 Problem solving principles:

- Completely understand the problem
- Devise a plan to solve it
- Carry out the plan
- Review the results

2.2.7 Developing a Program:

- Analyze the problem
- Design the program
- Code the program
- Test the program

3 What is a CPU?

In the 19th century, Babbage made advances in mechanical computers. In 1930 electromechanical computers were developed by Konrad Zuse in Germany. By World War II, warships had mechanical computers in the form of fire control systems. The first digital computer was built at Iowa State slightly before WWII. After World War II, the Americans built a digital computer using vacuum tubes for military ballistic studies. Like many post WWII technologies, computers were originally created for military purposes; however, the real advances did not materialize until they were applied to business problems thereby creating a large demand, which promoted advances through economic competition.

[6]:
```
Image(filename="babbage.png")
```

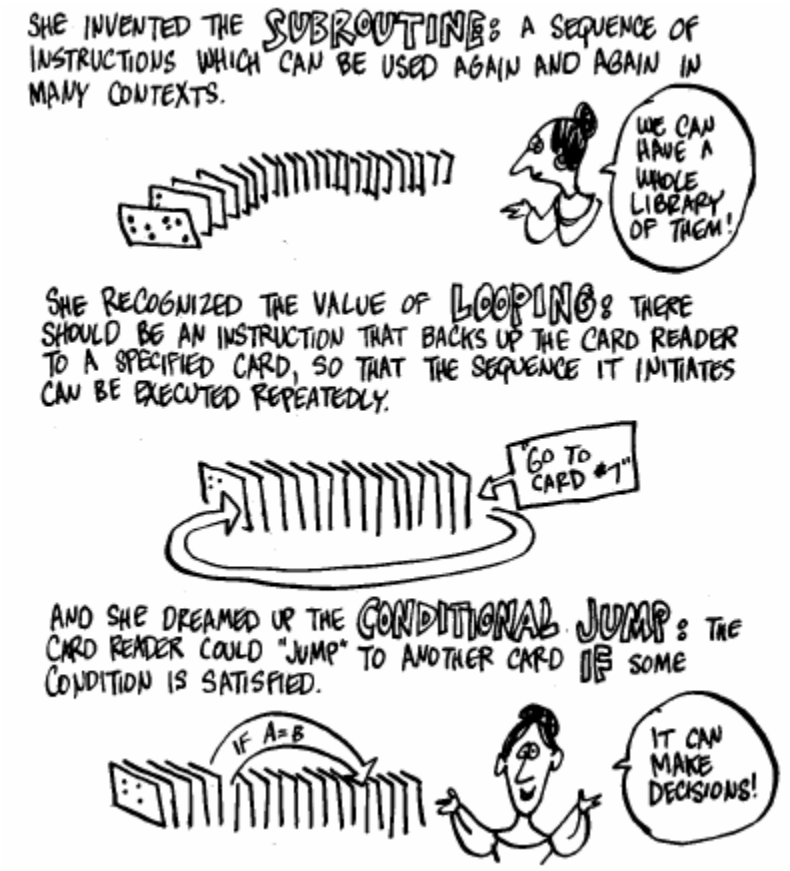
[6]:

Features:
- Processing unit (the Mill) - Memory (the Store) - Programmable (punched cards)
- Iteration, conditional branching, pipelining, many I/O devices

The first programmer was a girl:

[7]:
```
Image(filename="ada.png")
```

[7]:



Advances in computers are made possible by advances in microelectronics. This is because (1) mechanical computers are far too large and too inaccurate, and (2) vacuum tube computers consume a great deal of power and break down frequently. What made the computer a commercial success was the invention of an inexpensive, reliable transistor, which required very little power. At first, transistors in computers were individually wired components. Technological advance led to boards with individual components, and finally to boards with integrated circuits. The advance in microelectronics has made the digital computer dominant over other types of computers because of its lower cost and higher performance.

In a multi socket (CPU) design (workstation, server, supercomputer node) each usually CPU replicates a cache and memory layout as above and is connected to other CPUs on the same board by QPI or HT links.

4 What is Python?

A versatile language focused on the ease of development. Created by Guido van Rossum in 1989. Its development and promotion is managed by the Python Software Foundation (PSF) since 2001.

Its name comes from the television series ``Monty Python's Flying Circus''.

It was primarily intended for teaching programming languages, so much emphasis was given in the simplicity of the syntax. Python is a high-level (semi-)interpreted language.

4.0.1 High level and low level languages

Python is a *high level language*: its code is mean to be read by humans (or things pretending to be). Other high level languages include *C, C++, Fortran, Java, R, Perl* and many more.

Computers however ``do not understand'' i.e. they cannot directly execute programs in a high level language. These have be translated in *machine* languages which are specific to a definite type or family of processors (an *architecture*).

Thus, using high level languages is beneficial in terms of *readability* but also of *portability*.

A program in a high level language may be converted in assembly before being ever executed; this is done a something called *a compiler*. In this context the high level program is called **the source code** and the compiled one is an **object file** (it may be or not directly executable).

In alternative, the high level program is translated *on the fly* by a program called *an interpreter* reading a file or in an interactive environment (like the one we are using). In this context one talks of **scripts**.

Some fundamentals features

1. Dynamic typing: the type of a variable is set from its content, the way it is used and the methods which can be applied to it (with restrictions)
2. Provides automatic memory management.
3. Emphasizes code readability, in particular by forcing code indentation define block structures.
4. Case-sensitive.

Duck Typing: When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck. • The type of a variable is based on its methods and properties and not explicitly provided.

• An important feature of duck typing is that variables with different properties can still be used in the same sequence of instructions provided they have the properties and methods needed in this sequence.

• A consequence of this feature is that procedures can be simply used with arguments of different types if they support the instructions present in the procedure.

** Python is actually a language definition. It standard or reference implementation is cPython**
But other flavours exists, such as:

1. PyPy

2. Jython
3. IronPython

4.0.2 Is anybody really using it?

Google

YouTube

DropBox

Wikipedia

Open-Office

Yahoo

Sito web Nasa.org

BitTorrent

4.0.3 The Hagakure:

1 -- Comments are free, like a free beer. By comparison, they have to be a glorious thing

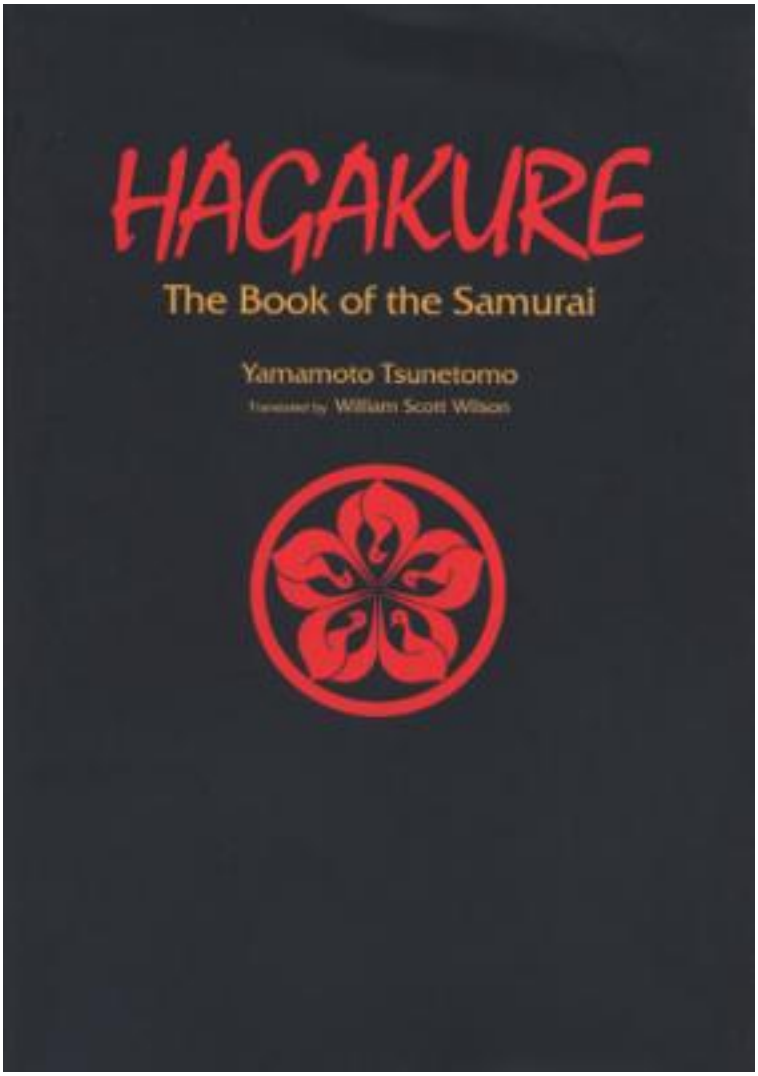
2 - Programmer time is expensive; conserve it in preference to machine time.

3 - Write a big program only when it is clear by demonstration that nothing else will do.

4 - Distrust all claims for ``one true way''

[8]:
```
Image(filename="Hagakure.jpg")
```

[8]:



4.0.4 Python 2 vs Python 3

Short version:

Python 2.x is the status quo; many big scientific and non scientific projects are written in Python 2.x

Python 3.x is the present and future of the language; intentionally cleared up by Guido Van Rossum without much worrying about back-porting

We will use Python 3.4 or 3.5

Conversion from 2.x to 3.x is not that difficult and many case can be semiautomatic. See also Section ??

4.0.5 Notebooks

Notebooks have code cells (often followed by result cells) and text cells. The text cells are the stuff that you're reading now. The code cells start with ``In []:'' with some number generally in the brackets. If you put your cursor in the code cell and hit Shift-Enter, the code will run in the Python interpreter and the result will print out in the output cell. You can then change things around and see whether you understand what's going on.

All course material is presented in notebooks; solutions to exercises are provided in separate notebooks. Notebooks will be uploaded to the SNS Google Drive just before the lesson.

4.0.6 Python as a calculator

[9]:
```
2+2*10
```

[9]: 22

[10]:
```
(2+2.5)*10
```

[10]: 45.0

[11]:
```
2**3
```

[11]: 8

Operator precedence and nesting works as expected. Division needs some caveats:

[12]:
```
2/3
```

[12]: 0.6666666666666666

[13]:
```
2//3
```

[13]: 0

[14]:
```
int(2/3)
```

[14]: 0

Python 2.x works differently, integer division is default.
Other mathematical functions are available using *math* (see below for more about import)

[15]:
```
import math
print(math.sqrt(9))
math.sin(1.7)
```

3.0

[15]: 0.9916648104524686

5 Python types

5.0.1 Numeric types

Integer: Integers in Python are represented on unsigned 32 or 64 bits by default (depending on the interpreter). The default base is 10 but bases 2 (0b…), 8 (0o…) and 16 (0x…) are also supported.

Long integers have been deprecated from Python 2.x

[16]:
```
0b100001
```

[16]: 33

[17]:
```
0o41
```

[17]: 33

[18]:
```
0x21
```

[18]: 33

Float: Approximate representation of a real on 64 bits (double precision in the IEEE-754 norm). `` e '' (case-insensitive) can be used to represent the exponential form, read as ``times 10 to the power''. If the decimal power is 0, the exponent character can be omitted.

[19]:
```
10e2
```

[19]: 1000.0

Complex: Approximate representation of a complex. The real and imaginary parts are each stored on 64 bits. The real and imaginary parts are given as a sum with the imaginary part identified with `` j '' placed next to the number

[20]:
```
1.0 + 5.0j
```

[20]: (1+5j)

Arithmetic operators:

\- * / + ** % abs

Comparison:

< <= > >= == !=

string: Represents a character string of at least one character, delimited by single or double quotes (' or "). Multi-line strings are given with tripled delimiters (''' or """). As of Python 3, all strings are in unicode.

boolean: Logical constants: True and False . All previous types have an associated boolean value: 0 , '' (empty string) are considered *False*, the other ones *True*.

None: Used to declare the absence of values. It can be used to initialize values. The associated boolean value is *False* (see below).

[21]:
```
True + 1
```

[21]: 2

[29]:
```
"" == True
```

[29]: False

[23]:
```
3>3.142
```

[23]: False

[24]:
```
3!=2
```

[24]: True

The += and -= operators are unary and can increase or decrease a variable:

[25]:
```
a=10; a+=3; a
```

[25]: 13

[26]:
```
a-=4; a
```

[26]: 9

6 Hello, World!

[27]:
```
myvar="Hello World!"
#this is comment; remember, IT'S FREE
print(myvar)
#notice that the print function prints
myvar="the looooooooooong \
goodbye"
print("Hello","World",sep="+++")
myvar
```

```
#this is another comment, just to nail it
```

```
Hello World!
Hello+++World
```

[27]:
```
'the looooooooooong goodbye'
```

Note that above we defined *myvar* as a variable. You can name a variable *almost* anything you want. It needs to start with an alphabetical character or ``_'', can contain alphanumeric characters plus underscores (``_''). Certain words, however, are reserved for the language:

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

Trying to define a variable using one of these will result in a syntax error:

[31]:
```
continue=10
```

```
  File "/tmp/ipykernel_231330/970167645.py", line 2
    continue=10
             ^
SyntaxError: invalid syntax
```

Python defines blocks using indentation levels, made by either tabs or spaces (better). When the interpreter finds a starting space or tab it assumes you are defining a block

[32]:
```
print(myvar)
  print(10)
```

```
  File "/tmp/ipykernel_231330/587829432.py", line 2
    print(10)
    ^
IndentationError: unexpected indent
```

input function:

[79]:
```
a=input("Enter a string: ")
```

```
Enter a string: Jack Lamotta
```

[80]:
```
if a == "Jack Lamotta":
    print("the Raging Bull")
elif a != None:
    print("non empty string")
else:
    print("empty string")
```

```
the Raging Bull
```

How long is string?

[81]:
```
len("qwerty")
```

[81]:
```
6
```

6.0.1 String Methods

S.capitalize(), S.lower(), S.upper(), S.replace()
S.isalpha(), S.isdigit(), S.islower(), S.isspace(), S.isupper(), s.count()
S.join(sequence)
S.startswith(), S.endswith(), S.find()

[82]:
```
mystring="snake plinskeen"
mystring.capitalize()
```

[82]:
```
'Snake plinskeen'
```

[83]:
```
mystring.replace(" ","++")
```

[83]:
```
'snake++plinskeen'
```

6.0.2 Strings and sequences

Note that strings supports simple arithmetic operations such as addition and multiplication:

[84]:
```
"qwe" + "rty"
```

[84]:
```
'qwerty'
```

[85]:
```
"qwe"*3
```

[85]:
```
'qweqweqwe'
```

string are sequence of characters; this allows to refer to a specific character along a string using the following syntax:

[86]:
```
mystring="snake plinskeen"
#to store character in position 5 of mystring
```

```
char5 = mystring[4]
char5
```

[86]: 'e'

** we start counting from 0!**

7 Flow control

7.1 Iteration, Indentation, and Blocks

[87]:
```
buzzwords=["foo","buzz","bar"]
for word in buzzwords:
    UP = word.capitalize()
    print(UP)
```

```
Foo
Buzz
Bar
```

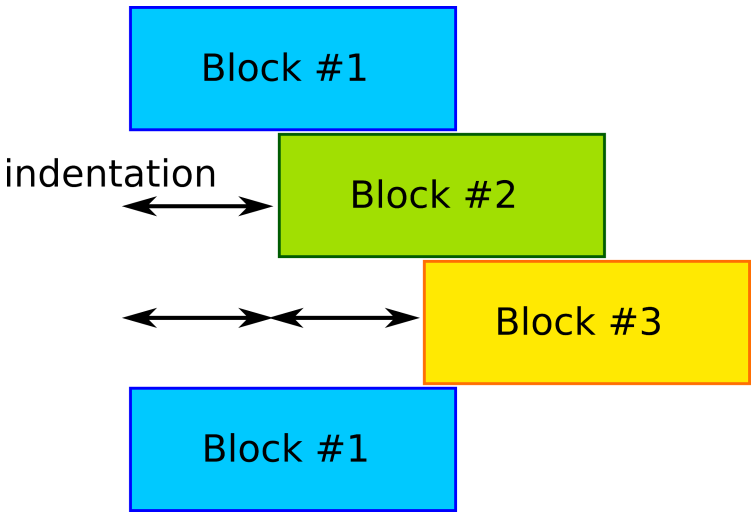
The code snippet above goes through each element of the *list* (see below) called **buzzwords** and assigns it to the variable **word**. It then executes everything in the indented block (in this case only one line of code, the print statement) using those variable assignments. When the program has gone through every element of the list, it exists the block.

(Almost) every programming language defines blocks of code in some way. In Fortran, one uses END statements (ENDDO, ENDIF, etc.) to define code blocks. In C, C++, and Perl, one uses curly braces {} to define these blocks.

Python uses a colon (``:''), followed by indentation level to define code blocks. Everything at a higher level of indentation is taken to be in the same block. In the above example the block was only a single line, but we could have had longer blocks as well:

[88]:
```
Image(filename="blocks.png")
```

[88]:



Code blocks are defined by indentation levels, created by a fixed abount of *whitespace* (usually 4 or 8) or <Tab>. As Linus says:

First off, I'd suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it's a great symbolic gesture.

Anyway, here goes:

    Chapter 1: Indentation

Tabs are 8 characters, and thus indentations are also 8 characters. There are heretic movements that try to make indentations 4 (or even 2!) characters deep, and that is akin to trying to define the value of PI to be 3.

Rationale: The whole idea behind indentation is to clearly define where a block of control starts and ends. Especially when you've been looking at your screen for 20 straight hours, you'll find it a lot easier to see how the indentation works if you have large indentations.

Now, some people will claim that having 8-character indentations makes the code move too far to the right, and makes it hard to read on a 80-character terminal screen. The answer to that is that if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.

Loops in Python are written as:

```
for item in iterable:
    body

for var in range-of-values:
    body

while condition:
    body
```

[89]:
```
count=1
while count<=5:
    print("counting ",count) #list of items to print
    count += 1
```

```
counting  1
counting  2
counting  3
counting  4
counting  5
```

7.2 Conditional execution

We invariably need some concept of conditions in programming to control branching behavior, to allow a program to react differently to different situations. If it's Monday, I'll go to work, but if it's Sunday, I'll sleep in. To do this in Python, we use a combination of boolean variables, which evaluate to either True or False, and if statements, that control branching based on boolean values.

```
if condition :
    body
elif condition :
    body
else:
    body
```

[90]:
```
x=11
y="q"
if x%2 == 0:
    y = y * x
else:
    y = y + str(x)
print(y)
```

```
q11
```

Loops may be stopped with the **break** statement; single iterations may be skipped with **continue**

[91]:
```
for i in range(10):
    if i < 3:
        continue
    elif i > 8:
        break
    print(i)
```

```
3
4
5
6
```

```
7
8
```

[92]:
```
for i in range(5): print(i)
```

```
0
1
2
3
4
```

[93]:
```
day="Sunday"
if day == "Sunday":
    print("Sleep in")
else:
    print("Go to work")
```

```
Sleep in
```

The ``=='' operator performs equality testing. If the two items are equal, it returns True, otherwise it returns False. In this case, it is comparing two variables, t

[94]:
```
day == "Sunday"
```

[94]: True

There is another boolean operator **is**, that tests whether two objects are the same object:

[95]:
```
day == "Sunday"
```

[95]: True

[96]:
```
day == "Monday"
```

[96]: False

If we evaluate it by itself, as we just did, we see that it returns a boolean value, False. The if statement that contains the truth test is followed by a code block (a colon followed by an indented block of code). If the boolean is true, it executes the code in that block. Since it is false in the above example, we don't see that code executed.

[97]:
```
1>=2
```

[97]: False

[98]:
```
1!=2
```

[98]: True

[99]:
```
1==1.0
```

[99]: True

[101]:
```
1 == 1.0
```

[101]: True

1 == 1.0 test is true: even if the two objects are different data types (integer and floating point number), they have the same *value*.

[102]:
```
a=1
a == True
```

[102]: True

[103]:
```
a=0
a == True
```

[103]: False

[104]:
```
a=None
a == True
```

[104]: False

[105]:
```
bool("q")
```

[105]: True

The Fibonacci sequence is a sequence in math that starts with 0 and 1, and then each successive entry is the sum of the previous two. Thus, the sequence goes 0,1,1,2,3,5,8,13,21,34,55,89,…

A very common exercise in programming books is to compute the Fibonacci sequence up to some number **n**. First I'll show the code, then I'll discuss what it is doing.

[106]:
```
length = 6
sequence = [0,1] #another list, like buzzwords
for i in range(2,length):
    if length<=2: # control
        break
    sequence.append(sequence[i-1]+sequence[i-2])
print(sequence)
```

[0, 1, 1, 2, 3, 5]

Let's go through this line by line. First, we define the variable **length**, and set it to the integer **6**. We then create a variable called **sequence**, and initialize it to the list with the integers 0 and 1 in it, the first two elements of the Fibonacci sequence. We have to create these elements ``by hand'', since the iterative part of the sequence requires two previous elements.

In the body of the loop, we append to the list an integer equal to the sum of the two previous elements of the list.

After exiting the loop (ending the indentation) we then print out the whole list. That's it!

We might want to use the Fibonacci snippet with different sequence lengths. We could cut an paste the code into another cell, changing the value of **n**, but it's easier and more useful to make a **function** out of the code. We do this with the **def** statement in Python:

[107]:
```
def fibonacci(sequence_length):
    """
    Return the Fibonacci sequence of length *sequence_length*
    """
    #this is a docstring
    sequence = [0,1]
    if sequence_length < 1:
        print("Fibonacci sequence only defined for length 1 or greater")
        return
    if 0 < sequence_length < 3:
        return sequence[:sequence_length]
    for i in range(2,sequence_length):
        sequence.append(sequence[i-1]+sequence[i-2])
    return sequence
```

We can now call **fibonacci()** for different sequence_lengths:

[108]:
```
fibonacci(2)
```

[108]:
```
[0, 1]
```

[109]:
```
fibonacci(7)
```

[109]:
```
[0, 1, 1, 2, 3, 5, 8]
```

If you define a docstring for all of your functions, it makes it easier for other people to use them, since they can get help on the arguments and return values of the function:

Finally, note that rather than putting a comment in about what input values lead to errors, we have some testing of these values, followed by a warning if the value is invalid, and some conditional code to handle special cases.

7.3 Exercises

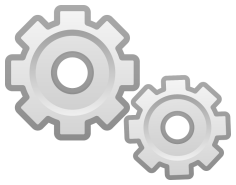
1. Write a right_justify function which, given a string as input, adds leading spaces and prints a 70 char. wide line:
2. Given a string s, return a string made of the first 2 and the last 2 chars of the original string, so `spring' yields `spng'. If the string length is less than 2, return an empty string.
3. Given strings a and b, return a single string with a and b separated by a space, except swap the first 2 chars of each string. Assume a and b are length 2 or more.

   'mix', pod' -> 'pox mid'

   'dog', 'dinner' -> 'dig donner'
4. Given a string, if its length is at least 3, add `ing' to its end. Unless it already ends in `ing', in which case add `ly' instead. If the string length is less than 3, leave it unchanged. Return the resulting string.
5. Consider dividing a string into two halves. If the length is even, the front and back halves are the same length. If the length is odd, we'll say that the extra char goes in the front half. e.g. `abcde', the front half is `abc', the back half `de'. Given 2 strings, a and b, return a string of the form a-front + b-front + a-back + b-back

[110]:
```
Image(filename="gears.png")
```

[110]:



7.4 Solutions

[111]:
```
# ex. 1
def right_justify(string):
    L = len(string)
    if L >= 70:
        return string
    return " "*(70-L) + string
right_justify("Lee van Cleef")
```

[111]:
```
'                                                                     Lee van Cleef'
```

[112]:
```
#ex. 2
def both_ends(s):
    if len(s) < 2:
        return '' #we use return to immediatly exit the function
    first2 = s[0:2]
    last2 = s[-2:]
    return first2 + last2
both_ends("plinskeen")
```

[112]: 'plen'

[113]:
```
def mix_up(a, b):
    a_swapped = b[:2] + a[2:]
    b_swapped = a[:2] + b[2:]
    return a_swapped + ' ' + b_swapped
mix_up("snake","plinskeen")
```

[113]: 'plake sninskeen'

[114]:
```
def verbing(s):
    if len(s) >= 3:
        if s[-3:] != 'ing': s = s + 'ing'
        else: s = s + 'ly'
    return s
verbing("punch")
```

[114]: 'punching'

[115]:
```
def front_back(a, b):
    a_middle = len(a) // 2
    b_middle = len(b) // 2
    if len(a) % 2 == 1:  # add 1 if length is odd
        a_middle = a_middle + 1
    if len(b) % 2 == 1:
        b_middle = b_middle + 1
    return a[:a_middle] + b[:b_middle] + a[a_middle:] + b[b_middle:]
front_back("The","Warriors")
```

[115]: 'ThWarreiors'

8 Interactive and batch mode

Interactive sessions are useful for testing and/or for educational purposes and IPython provides an excellent interactive and *reusable* environment. However, code is usually saved in one or more plain ASCII files (Python scripts) and then read and executed by the interpreter in batch mode. The interpreter is called from the command line or starting the script with #!/usr/bin/env python3:

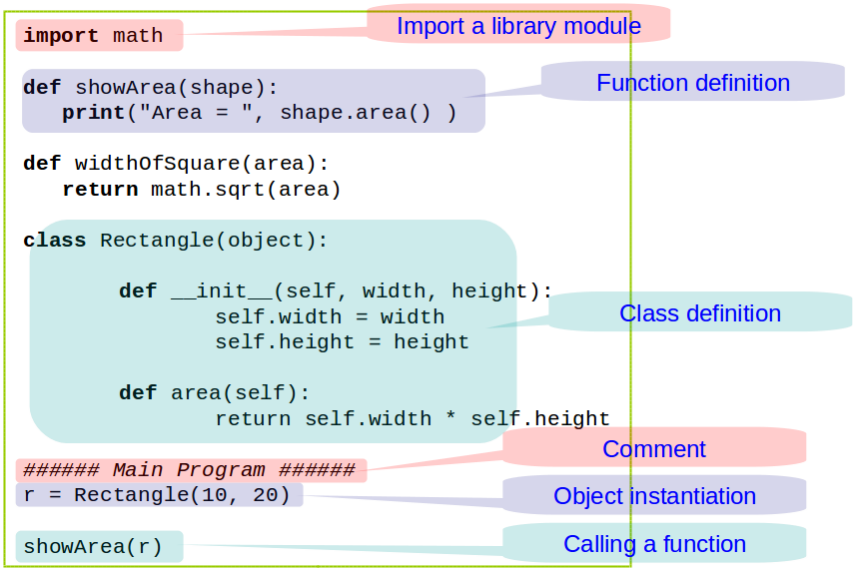
```
[g.mancini@shangrila ~]$ echo 'print("Hello","\n","World!")' >> hello.py

[g.mancini@shangrila ~]$ python3 hello.py
    Hello
     World!
[g.mancini@shangrila ~]$ rm -f hello.py
[g.mancini@shangrila ~]$ echo  '#!/usr/local/bin/python3' > hello.py
[g.mancini@shangrila ~]$ echo 'print("Hello World")' >>  hello.py;
[g.mancini@shangrila ~]$ cat myscript.py ; chmod u+x hello.py
#!/usr/local/bin/python3
print("Hello World")
[g.mancini@shangrila ~]$ ./hello.py
Hello World
[g.mancini@shangrila ~]$
```

8.0.1 Structure of a Python Script

[116]:
```
Image(filename="structure.png")
```

[116]:



IPython provides the magic command %run to allow execution of scripts in the notebook:

[117]:
```
%run hello.py
```

```
␣
↪--------------------------------------------------------------------------
```

```
NameError                                 Traceback (most recent call last)

~/Dropbox/teach/Medicinal_phys_chem_2022/01_Introduction_to_python/hello.py in <module>
----> 1 print("Hello","World","!",myvar)


NameError: name 'myvar' is not defined
```

Variables defined in the notebook environment can be imported into the script by running %run -i:

[118]:
```
!cat hello.py
```

```
print("Hello","World","!",myvar)
```

[119]:
```
myvar="my variable"
%run -i hello.py
```

```
Hello World ! my variable
```

Notice how above we invoked the Linux BASH shell with !.

Other necessary IPython magic commands include (many more available):

```
%reset:  clear all namespace
%hist:   print history
%xdel:   delete variables
%who:   list objects in enviroment (use in combo with ? to know who is who)
```

9 Modules

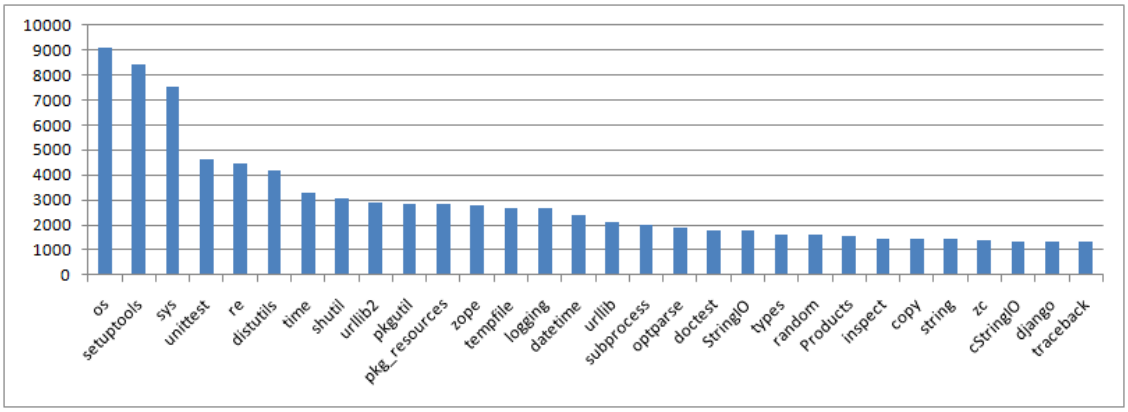
1. Use classes & functions defined in another file
2. A Python module is a file with the same name (plus the .py extension)
3. Like Java import, C/C++ include
4. Three formats of the command: import somefile from somefile import * from somefile import className

The difference? What gets imported from the file and what name refers to it after importing. A standard installation of Python usually comes with more than 200 modules available. Number of modules available online is about 10000 (and growing).

Here are the top 30 ``base modules'', ordered by number of PyPI projects importing them. These results are based on 11,204 packages download from PyPI.

[120]:
```
Image(filename="module_stats.png")
```

[120]:



Section ??

import somefile

Everything in somefile.py gets imported. To refer to something in the file, append the text ``somefile.'' to the front of its name.

from somefile import *

Everything in somefile.py gets imported. To refer to anything in the module, just use its name. Everything in the module is now in the current namespace.

NB Using this import command can easily overwrite the definition of an existing function or variable!

from somefile import className

Only the item *className* in somefile.py gets imported. After importing *className*, you can just use it without a module prefix. It's brought into the current namespace. NB Overwrites the definition of this name if already defined in the current namespace!

[121]:
```
import sys
sys.argv
```

[121]:
```
['/usr/local/lib/python3.8/dist-packages/ipykernel_launcher.py',
 '-f',
 '/home/gmancini/.local/share/jupyter/runtime/kernel-c2e29cd1-cdf1-48de-8339-2a6
2460e10ec.json']
```

[122]:
```
argv
```

```
␣
↪---------------------------------------------------------------------------
```

```
        NameError                                 Traceback (most recent call␣
↪last)

        ~/Dropbox/teach/Medicinal_phys_chem_2022/01_Introduction_to_python/hello.
↪py in <module>
   ----> 1 argv


        NameError: name 'argv' is not defined
```

sys.argv provides a very simple way of accetping command line arguments:

[123]:
```
%cd /home/gmancini
!ls
%cd -
```

```
/home/gmancini
archive    data       Downloads  Music     pkg     Templates
Books      Desktop    Dropbox    old-work  Public  Videos
Corso_Kub  Documents  ISOim      Pictures  snap    Zotero
/home/gmancini/Dropbox/teach/Medicinal_phys_chem_2022/01_Introduction_to_python
```

[124]:
```
!pwd
```

```
/home/gmancini/Dropbox/teach/Medicinal_phys_chem_2022/01_Introduction_to_python
```

[125]:
```
!cat args.py
```

```
#!/usr/bin/env python3

from sys import argv

def dumb_function():
    for num,arg in enumerate(argv):
        print(num, arg)

dumb_function()
```

[126]:
```
%run args "foo" "bar"
```

```
0 args.py
1 foo
2 bar
```

Import everything in a module in the current namespace

[127]:
```
from re import *
findall
```

[127]:
```
<function re.findall(pattern, string, flags=0)>
```

the interpreter tells you that findall is a function

The *help()* and *dir()* builtins allow you to check what you have included so far and how to use it (notice the __ of modules imported by default):

[128]:
```
dir()[-10:] #we use slicing, wait until the next section
```

[128]:
```
['sequence',
 'split',
 'sub',
 'subn',
 'sys',
 'template',
 'verbing',
 'word',
 'x',
 'y']
```

[129]:
```
help(math.floor)
```

```
Help on built-in function floor in module math:

floor(x, /)
    Return the floor of x as an Integral.

    This is the largest integer <= x.
```

IPython supports Tab-completion and **introspection**:

[130]:
```
?myvar
```

You can use import to use a function defined in a script:

[131]:
```
!ls
```

```
01_What_Is_Python.ipynb  gears.png      module_stats.png       tacc1.png
ada.png                  Hagakure.jpg   moore.png              tacc2.png
args.py                  hamming.png    precision.png          userspace.png
assembly.png             hello.py       python2python3.pdf     vis.png
avogadro.png             IBM.png        representation.png     viswork.png
babbage.png              languages.png  structure.png
blocks.png               lan_hie.png    Supercomputers-OS.png
coords.png               means.png      SVG
```

[132]:
```
import args
dumb_function()
```

```
0 /usr/local/lib/python3.8/dist-packages/ipykernel_launcher.py
1 -f
2 /home/gmancini/.local/share/jupyter/runtime/kernel-c2e29cd1-cdf1-48de-8339-2a6
2460e10ec.json
0 args.py
1 foo
2 bar
```

10 File manipulation

To open a file:

open(filename, mode)

mode is one of: `r' (Read), `w' (Write), `a' (Append). If a file opened for `w' does not exist it will be created.

Common methods for file handles include:

f.readline(): read a single line from a file. Returns a string.
f.readlines([size]): read all the lines up to size bytes and return them in a iterator of stri
f.read([size]): read up to size bytes; returns a string
f.write(text): write text to file

[133]:
```
%%bash
awk '{print "foo","baz","bar"}' > foo.txt
```

[134]:
```
f = open('foo.txt')
```

[135]:
```
f = open('foo.txt',"w")
f.write("Dutch Dillon Billy")
f.close()
```

[136]:
```
!ls -rt
```

```
blocks.png      module_stats.png  gears.png               vis.png
babbage.png     means.png         coords.png              viswork.png
avogadro.png    lan_hie.png       Supercomputers-OS.png   SVG
assembly.png    languages.png     structure.png           representation.png
args.py         IBM.png           python2python3.pdf      01_What_Is_Python.ipynb
ada.png         hello.py          tacc1.png               __pycache__
precision.png   hamming.png       tacc2.png               foo.txt
moore.png       Hagakure.jpg      userspace.png
```

[137]:
```
!cat foo.txt
```

```
Dutch Dillon Billy
```

[138]:
```
f = open('foo.txt')
f.readline()
```

[138]:
```
'Dutch Dillon Billy'
```

[139]:
```
!rm -f foo.txt
```

[140]:
```
!ls -rt
```

```
blocks.png      module_stats.png  gears.png               vis.png
babbage.png     means.png         coords.png              viswork.png
avogadro.png    lan_hie.png       Supercomputers-OS.png   SVG
assembly.png    languages.png     structure.png           representation.png
args.py         IBM.png           python2python3.pdf      01_What_Is_Python.ipynb
ada.png         hello.py          tacc1.png               __pycache__
precision.png   hamming.png       tacc2.png
moore.png       Hagakure.jpg      userspace.png
```

If handling a text file, there are more useful functions such f.readlines

11 Exercises

1. Read the *readlines()* documentation. Write a script that counts the number of lines in this notebook
2. Think about the difference between !, % and %% magics
3. Write a script that copies a text file into another taking filenames as arguments

12 Installing Python

12.1 Windows

(disclaimer: in the past 13 years I *seldom* used windows)

Simply donwnload the installer from https://www.python.org/downloads/. Additional modules (Numpy, Ipython) can be download and installed normally.

Alternatively, you can simply install *Anaconda* from continuum analysics (https://www.continuum.io/downloads, never used myself, available also for OS X and Linux) which provides a useful a complete environment (in the words of most users).

Canopy (https://store.enthought.com/downloads/#default) is another multiplatform alternative.

12.2 Linux / OSX

(c)Python 2.x is shipped by default on major Linux distributions; you can use **pip** to install additional packages.

Python3 is often an optional package that can be installed using the distro's or osx package manager (*yum, apt, yast, pacman, Homebrew, …*) or compiled from the source code.

12.3 Developing with Python

IPython and a text editor are often more than enough, but if you want a complete IDE, *eclipse* and *NetBeans* are available. On Linux the *Kate* editor, among others, has extension for syntax checking and code completition.

Editor specifically developed for (and with) Python are also available such as:

1. Spyder
2. Dr Python
3. Komodo
4. Eric

12.3.1 Installing modules

Python 3.4 and and Python 2.7.9 now ship with the pip package manager, allowing easy install of additional modules: sudo pip3 install mdanalysis when using the import keyword, python searches for modules in the current directory, where Python executables are located and in the PYTHONPATH environment variable: export PYTHONPATH=

[141]:
```
import sys
sys.path
```

[141]:
```
['/home/gmancini/Dropbox/teach/Medicinal_phys_chem_2022/01_Introduction_to_pytho
n',
 '/usr/lib/python38.zip',
 '/usr/lib/python3.8',
 '/usr/lib/python3.8/lib-dynload',
 '',
 '/usr/local/lib/python3.8/dist-packages',
 '/usr/lib/python3/dist-packages',
 '/usr/local/lib/python3.8/dist-packages/IPython/extensions',
 '/home/gmancini/.ipython']
```

Previously, we have used the math and sys modules. Another very useful module is os which module provides a portable way of using operating system dependent functionality:

[142]:
```
import os
dir(os)
```

[142]:
```
['CLD_CONTINUED',
 'CLD_DUMPED',
 'CLD_EXITED',
 'CLD_TRAPPED',
```

```
'DirEntry',
'EX_CANTCREAT',
'EX_CONFIG',
'EX_DATAERR',
'EX_IOERR',
'EX_NOHOST',
'EX_NOINPUT',
'EX_NOPERM',
'EX_NOUSER',
'EX_OK',
'EX_OSERR',
'EX_OSFILE',
'EX_PROTOCOL',
'EX_SOFTWARE',
'EX_TEMPFAIL',
'EX_UNAVAILABLE',
'EX_USAGE',
'F_LOCK',
'F_OK',
'F_TEST',
'F_TLOCK',
'F_ULOCK',
'GRND_NONBLOCK',
'GRND_RANDOM',
'MFD_ALLOW_SEALING',
'MFD_CLOEXEC',
'MFD_HUGETLB',
'MFD_HUGE_16GB',
'MFD_HUGE_16MB',
'MFD_HUGE_1GB',
'MFD_HUGE_1MB',
'MFD_HUGE_256MB',
'MFD_HUGE_2GB',
'MFD_HUGE_2MB',
'MFD_HUGE_32MB',
'MFD_HUGE_512KB',
'MFD_HUGE_512MB',
'MFD_HUGE_64KB',
'MFD_HUGE_8MB',
'MFD_HUGE_MASK',
'MFD_HUGE_SHIFT',
'MutableMapping',
'NGROUPS_MAX',
'O_ACCMODE',
'O_APPEND',
'O_ASYNC',
'O_CLOEXEC',
```

```
'O_CREAT',
'O_DIRECT',
'O_DIRECTORY',
'O_DSYNC',
'O_EXCL',
'O_LARGEFILE',
'O_NDELAY',
'O_NOATIME',
'O_NOCTTY',
'O_NOFOLLOW',
'O_NONBLOCK',
'O_PATH',
'O_RDONLY',
'O_RDWR',
'O_RSYNC',
'O_SYNC',
'O_TMPFILE',
'O_TRUNC',
'O_WRONLY',
'POSIX_FADV_DONTNEED',
'POSIX_FADV_NOREUSE',
'POSIX_FADV_NORMAL',
'POSIX_FADV_RANDOM',
'POSIX_FADV_SEQUENTIAL',
'POSIX_FADV_WILLNEED',
'POSIX_SPAWN_CLOSE',
'POSIX_SPAWN_DUP2',
'POSIX_SPAWN_OPEN',
'PRIO_PGRP',
'PRIO_PROCESS',
'PRIO_USER',
'P_ALL',
'P_NOWAIT',
'P_NOWAITO',
'P_PGID',
'P_PID',
'P_WAIT',
'PathLike',
'RTLD_DEEPBIND',
'RTLD_GLOBAL',
'RTLD_LAZY',
'RTLD_LOCAL',
'RTLD_NODELETE',
'RTLD_NOLOAD',
'RTLD_NOW',
'RWF_DSYNC',
'RWF_HIPRI',
```

```
'RWF_NOWAIT',
'RWF_SYNC',
'R_OK',
'SCHED_BATCH',
'SCHED_FIFO',
'SCHED_IDLE',
'SCHED_OTHER',
'SCHED_RESET_ON_FORK',
'SCHED_RR',
'SEEK_CUR',
'SEEK_DATA',
'SEEK_END',
'SEEK_HOLE',
'SEEK_SET',
'ST_APPEND',
'ST_MANDLOCK',
'ST_NOATIME',
'ST_NODEV',
'ST_NODIRATIME',
'ST_NOEXEC',
'ST_NOSUID',
'ST_RDONLY',
'ST_RELATIME',
'ST_SYNCHRONOUS',
'ST_WRITE',
'TMP_MAX',
'WCONTINUED',
'WCOREDUMP',
'WEXITED',
'WEXITSTATUS',
'WIFCONTINUED',
'WIFEXITED',
'WIFSIGNALED',
'WIFSTOPPED',
'WNOHANG',
'WNOWAIT',
'WSTOPPED',
'WSTOPSIG',
'WTERMSIG',
'WUNTRACED',
'W_OK',
'XATTR_CREATE',
'XATTR_REPLACE',
'XATTR_SIZE_MAX',
'X_OK',
'_Environ',
'__all__',
```

```
'__builtins__',
'__cached__',
'__doc__',
'__file__',
'__loader__',
'__name__',
'__package__',
'__spec__',
'_check_methods',
'_execvpe',
'_exists',
'_exit',
'_fspath',
'_fwalk',
'_get_exports_list',
'_putenv',
'_spawnvef',
'_unsetenv',
'_wrap_close',
'abc',
'abort',
'access',
'altsep',
'chdir',
'chmod',
'chown',
'chroot',
'close',
'closerange',
'confstr',
'confstr_names',
'copy_file_range',
'cpu_count',
'ctermid',
'curdir',
'defpath',
'device_encoding',
'devnull',
'dup',
'dup2',
'environ',
'environb',
'error',
'execl',
'execle',
'execlp',
'execlpe',
```

```
'execv',
'execve',
'execvp',
'execvpe',
'extsep',
'fchdir',
'fchmod',
'fchown',
'fdatasync',
'fdopen',
'fork',
'forkpty',
'fpathconf',
'fsdecode',
'fsencode',
'fspath',
'fstat',
'fstatvfs',
'fsync',
'ftruncate',
'fwalk',
'get_blocking',
'get_exec_path',
'get_inheritable',
'get_terminal_size',
'getcwd',
'getcwdb',
'getegid',
'getenv',
'getenvb',
'geteuid',
'getgid',
'getgrouplist',
'getgroups',
'getloadavg',
'getlogin',
'getpgid',
'getpgrp',
'getpid',
'getppid',
'getpriority',
'getrandom',
'getresgid',
'getresuid',
'getsid',
'getuid',
'getxattr',
```

```
'initgroups',
'isatty',
'kill',
'killpg',
'lchown',
'linesep',
'link',
'listdir',
'listxattr',
'lockf',
'lseek',
'lstat',
'major',
'makedev',
'makedirs',
'memfd_create',
'minor',
'mkdir',
'mkfifo',
'mknod',
'name',
'nice',
'open',
'openpty',
'pardir',
'path',
'pathconf',
'pathconf_names',
'pathsep',
'pipe',
'pipe2',
'popen',
'posix_fadvise',
'posix_fallocate',
'posix_spawn',
'posix_spawnp',
'pread',
'preadv',
'putenv',
'pwrite',
'pwritev',
'read',
'readlink',
'readv',
'register_at_fork',
'remove',
'removedirs',
```

```
'removexattr',
'rename',
'renames',
'replace',
'rmdir',
'scandir',
'sched_get_priority_max',
'sched_get_priority_min',
'sched_getaffinity',
'sched_getparam',
'sched_getscheduler',
'sched_param',
'sched_rr_get_interval',
'sched_setaffinity',
'sched_setparam',
'sched_setscheduler',
'sched_yield',
'sendfile',
'sep',
'set_blocking',
'set_inheritable',
'setegid',
'seteuid',
'setgid',
'setgroups',
'setpgid',
'setpgrp',
'setpriority',
'setregid',
'setresgid',
'setresuid',
'setreuid',
'setsid',
'setuid',
'setxattr',
'spawnl',
'spawnle',
'spawnlp',
'spawnlpe',
'spawnv',
'spawnve',
'spawnvp',
'spawnvpe',
'st',
'stat',
'stat_result',
'statvfs',
```

```
 'statvfs_result',
 'strerror',
 'supports_bytes_environ',
 'supports_dir_fd',
 'supports_effective_ids',
 'supports_fd',
 'supports_follow_symlinks',
 'symlink',
 'sync',
 'sys',
 'sysconf',
 'sysconf_names',
 'system',
 'tcgetpgrp',
 'tcsetpgrp',
 'terminal_size',
 'times',
 'times_result',
 'truncate',
 'ttyname',
 'umask',
 'uname',
 'uname_result',
 'unlink',
 'unsetenv',
 'urandom',
 'utime',
 'wait',
 'wait3',
 'wait4',
 'waitid',
 'waitid_result',
 'waitpid',
 'walk',
 'write',
 'writev']
```

[143]:
```
help(os.path)
```

```
Help on module posixpath:

NAME
    posixpath - Common operations on Posix pathnames.

MODULE REFERENCE
    https://docs.python.org/3.8/library/posixpath
```

```
    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    Instead of importing this module directly, import os and refer to
    this module as os.path.  The "os.path" name is an alias for this
    module on Posix systems; on other systems (e.g. Windows),
    os.path provides the same operations in a manner specific to that
    platform, and is an alias to another module (e.g. ntpath).

    Some of this can actually be useful on non-Posix systems too, e.g.
    for manipulation of the pathname component of URLs.

FUNCTIONS
    abspath(path)
        Return an absolute path.

    basename(p)
        Returns the final component of a pathname

    commonpath(paths)
        Given a sequence of path names, returns the longest common sub-path.

    commonprefix(m)
        Given a list of pathnames, returns the longest common leading component

    dirname(p)
        Returns the directory component of a pathname

    exists(path)
        Test whether a path exists.  Returns False for broken symbolic links

    expanduser(path)
        Expand ~ and ~user constructions.  If user or $HOME is unknown,
        do nothing.

    expandvars(path)
        Expand shell variables of form $var and ${var}.  Unknown variables
        are left unchanged.

    getatime(filename)
        Return the last access time of a file, reported by os.stat().

    getctime(filename)
        Return the metadata change time of a file, reported by os.stat().
```

```
getmtime(filename)
    Return the last modification time of a file, reported by os.stat().

getsize(filename)
    Return the size of a file, reported by os.stat().

isabs(s)
    Test whether a path is absolute

isdir(s)
    Return true if the pathname refers to an existing directory.

isfile(path)
    Test whether a path is a regular file

islink(path)
    Test whether a path is a symbolic link

ismount(path)
    Test whether a path is a mount point

join(a, *p)
    Join two or more pathname components, inserting '/' as needed.
    If any component is an absolute path, all previous path components
    will be discarded.  An empty last part will result in a path that
    ends with a separator.

lexists(path)
    Test whether a path exists.  Returns True for broken symbolic links

normcase(s)
    Normalize case of pathname.  Has no effect under Posix

normpath(path)
    Normalize path, eliminating double slashes, etc.

realpath(filename)
    Return the canonical path of the specified filename, eliminating any
    symbolic links encountered in the path.

relpath(path, start=None)
    Return a relative version of a path

samefile(f1, f2)
    Test whether two pathnames reference the same actual file or directory

    This is determined by the device number and i-node number and
```

```
        raises an exception if an os.stat() call on either pathname fails.

    sameopenfile(fp1, fp2)
        Test whether two open file objects reference the same file

    samestat(s1, s2)
        Test whether two stat buffers reference the same file

    split(p)
        Split a pathname.  Returns tuple "(head, tail)" where "tail" is
        everything after the final slash.  Either part may be empty.

    splitdrive(p)
        Split a pathname into drive and path. On Posix, drive is always
        empty.

    splitext(p)
        Split the extension from a pathname.

        Extension is everything from the last dot to the end, ignoring
        leading dots.  Returns "(root, ext)"; ext may be empty.

DATA
    __all__ = ['normcase', 'isabs', 'join', 'splitdrive', 'split', 'splite…
    altsep = None
    curdir = '.'
    defpath = '/bin:/usr/bin'
    devnull = '/dev/null'
    extsep = '.'
    pardir = '..'
    pathsep = ':'
    sep = '/'
    supports_unicode_filenames = False

FILE
    /usr/lib/python3.8/posixpath.py
```

13 References

These are the books I used to try to teach myself.

1. H. P. Langtaggen, *Python scripting for computational sciences*, Elsevier
2. M. Dawson, *Python programming for the absolute beginner*, Course Technology
3. M. Lutz, *Python Pocket Reference*, O'Reilly
4. D. Phillips *Python 3 Object-Oriented Programming - Third Edition*, Packt Publishing

many more (and better?) are available. However, the Internet is flooded with teaching material.

https://docs.python.org/2/tutorial/

www.python.org

are good places to start.

A proper referencing to examples, images and other stuff I used in the notebooks is still to be completed.

[144]:
```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```