Python 101
Modules

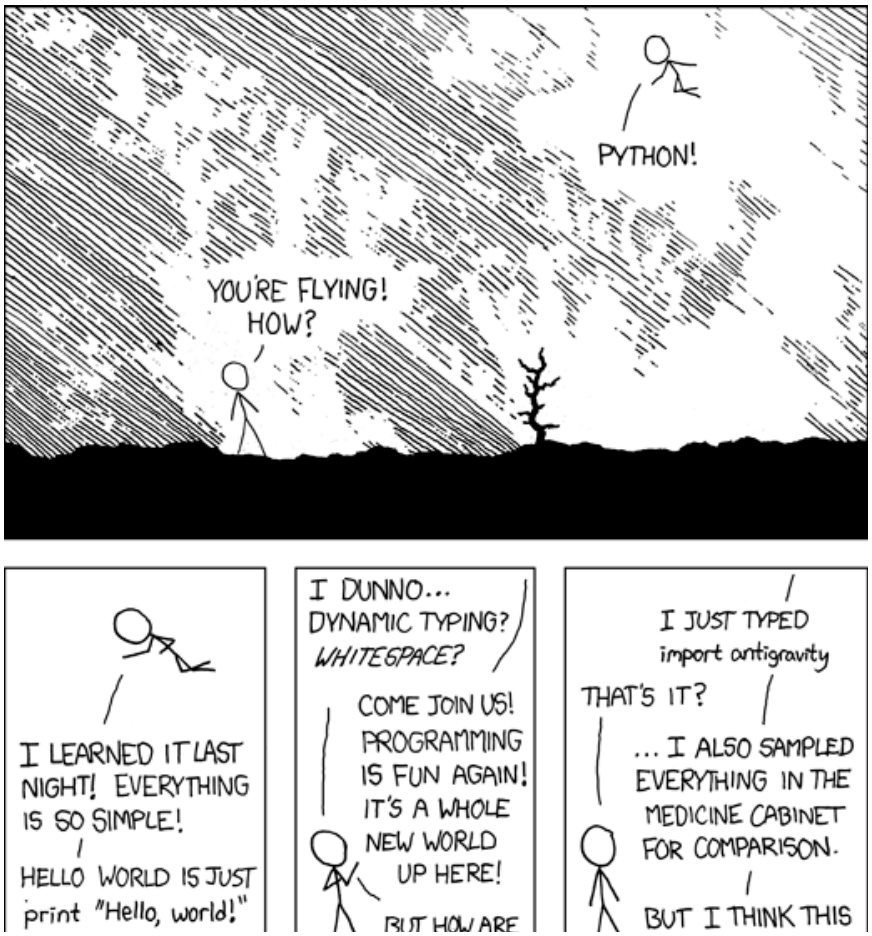
PYTHON 101 TEAM

Day Overview:

Today we will be speaking about python modules:

- What are modules?
- How do they work?
- Writing a module
- Some of the most popular/usefull modules
  - The ***sys*** (system) module
  - The ***re*** (regular expressions) module
  - The ***os*** (operating system) module
  - The ***subprocess*** (spawn new processes) module
  - The ***Bio*** (BioPython) module

What are modules?



What are modules?

```
 #Contents of module test.py
sequence_list = ["AGCTG","AGCTG","AGCTG"]

species_dictionary =
{"Hs":"Homo_sapiens","Mm":"Mus_musculus"}

def print_function (print_this):
    print print_this
```

- A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended.
- A module can contain statements as well as function definitions.
- Python has several *default* modules, which are present in any complete python install, such as ***re***, ***sys*** or ***os***. Consult here for a complete index

How do modules work?

Importing modules:

In order to use the contents of a specific module, you must use the import command.

```
import «module»
import «module» as «other_name»
from «module» import «something»
from «module» import «something» as «other_name»
from «module» import *
```

Using the contents of a module:

After importing a module, we can use it's contents in different ways depending on how

the module was imported:

```
import sys
print(sys.argv[0])

from sys import argv
print(argv[0])
```

Writing a module

In order to write a module, you just have to write a script that can be as simple as declaring some variables.

Afterwards, just import it into your *main* program and they will be ready for use.

However, it is a good practice, to add the following `if` statement to your code:

```
if __name__ == "__main__":
    «Program»
```

This will ensure that any code inside the conditional will **only** be run if the script is being run as a standalone program.

The sys module

The sys.argv function:

- Among many other advanced features, *sys* contains a very useful method: ***argv***. This method allows aditional arguments to be passed and used when invoking the script:

```
python my_script.py first_argument second_argument third_argument
```

- After the module has been imported into the script, the aditional arguments are stored in a list, which can be accessed in a simple way:

```
1  import sys
2  print str(sys.argv) # Note that the first argument in the
   list is the name of the script
4
```

```
["my_scripts.py","first_argument","second_argument","third_argumen
t"]
```

Calling external programs

```
os.popen(command[, mode[, bufsize]])
```

Deprecated since version 2.6: This function is obsolete. Use the subprocess module. Check especially the Replacing Older Functions with the subprocess Module section.

Python Documentation

The subprocess module

```
subprocess.check_output(args, *, stdin=None, stderr=None, shell=False, univers
al_newlines=False)
```

Run command with arguments and return its output as a byte string.

```
1  from subprocess import check_output
2  check_output(["echo", "Hello World!"])
4
```

```
'Hello World!\n'
```

The subprocess module

Shell pipe example

```
[bruno@laptop ~]$ ls -l | grep py
-rw-r--r-- 1 bruno cobig2      0 May 31 19:21 script1.py
-rw-r--r-- 1 bruno cobig2      0 May 31 19:21 script2.py
```

```
1  from subprocess import check_output
2  check_output("ls -l | grep py", shell=True)
3
5
```

```
-rw-r--r-- 1 bruno cobig2      0 May 31 19:21 script1.py
-rw-r--r-- 1 bruno cobig2      0 May 31 19:21 script2.py
```

More information

Python Documentation

The re module

The re module

About re:

- Regular expressions (RE) are a very large topic. A whole course could be had on them.
- They are very useful when our code starts getting full of endswith() and startswith() and lot's of conditionals all over.
- RE can make our lives a lot easier, but they take a lot of getting used to and even then, they produce hard to read code. But even despite these shortcomings, **they are awesome**!
- If you look here you can see that this section of python's documentation is as large

as say - the whole control flow section. But fear not, the docs are there to help you and you don't have to learn everything about RE today. The goal here is to let you know what can be done.

- Later you may want to go here to learn more. It is an introductory tutorial to RE.

The re module

What is a RE?

- A RE specifies a set of strings that match it; the functions in the *re* module let you check if a particular string matches a given regular expression.
- This often requires the use of *special* characters - AKA **metacharacters**.

The metacharacters

```
  .   -> Matches any character
  ^   -> Matches the beginning of a string (not a character)
  $   -> Matches the end of a string (also, not a character)
  *   -> Matches the preceeding character 0 or more times
  +   -> Matches the preceeding character 1 or more times
  ?   -> Matches the preceeding character 0 or 1 times
 {x}  -> Matches exactly _x_ copies of the preceeding character
{x,y} -> Matches _x_ to _y_ copies of the preceeding character
  \   -> Escapes the following character (for matching things like *)
[XYZ] -> Indicates a set of characters - in this case X, Y or Z
  |   -> Separates 2 or more REs, and matches either of them
```

There are, however, many more here

Using these *metacharacters* we can use the *re* module to perform useful operations, using re.search, re.sub and re.compile to name a few.

---

re.search

We will use re.search() as an example.

This function will look for an expression in a string and is invoked like this:

```
re.search(pattern, string, flags=0)
```

re.search will search a given string for a given pattern, and return it. If the pattern is not found, it returns *None*:

```
1  import re
2  test = "Python rules."
3  start = re.search("^.* ", test, flags=0)
4  print(start)
6
```

```
Python
```

You must test this code in IDLE or equivalent.

The os module

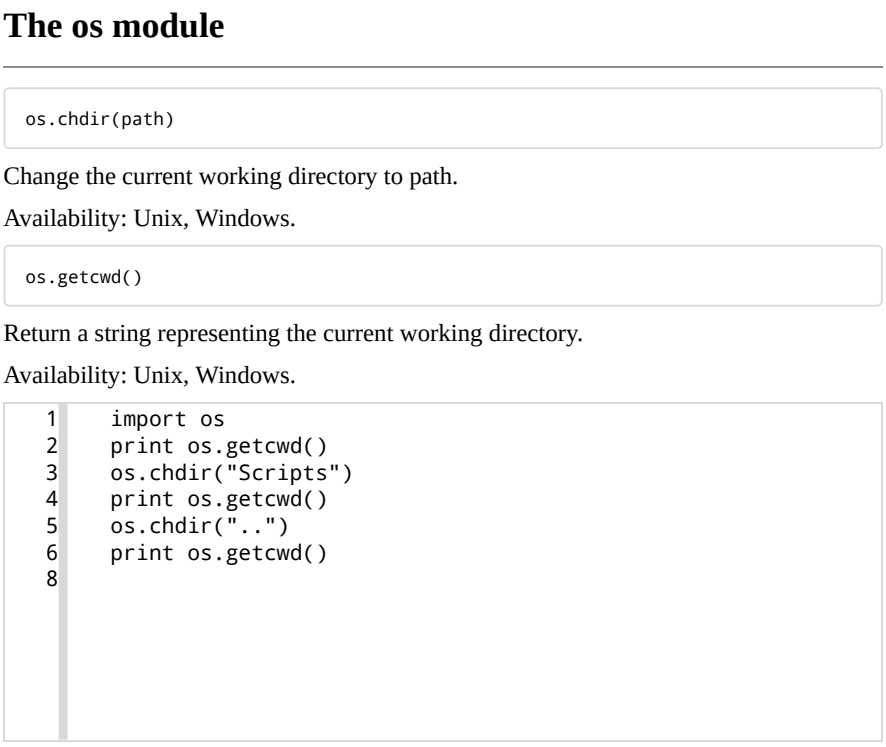
Miscellaneous operating system interfaces

Provides a portable way of using operating system dependent functionality.

- Some methods are only available on some OS
- Some methods return different results depending on the OS

More information

Python Documentation



The os module

```
os.chdir(path)
```

Change the current working directory to path.

Availability: Unix, Windows.

```
os.getcwd()
```

Return a string representing the current working directory.

Availability: Unix, Windows.

```
1  import os
2  print os.getcwd()
3  os.chdir("Scripts")
4  print os.getcwd()
5  os.chdir("..")
6  print os.getcwd()
8
```

```
'/home/bruno/Scripts'
'/home/bruno'
```

The os module

```
os.listdir(path)
```

Return a list containing the names of the entries in the directory given by path. The list is in arbitrary order.

Availability: Unix, Windows.

```
1  import os
2  print os.getcwd()
3  print os.listdir("Scripts")
4  print os.listdir(".")
5  print os.listdir("..")
7
```

```
'/home/bruno'
['script1.py', 'script2.py']
['Documents', 'Music', 'Movies', 'Scripts']
['bruno', 'diogo', 'francisco']
```

The os module

```
os.mkdir(path[, mode])
```

Create a directory named path. If the directory already exists, an error is raised.

Availability: Unix, Windows.

```
1  import os
2  print os.listdir(".")
3  os.mkdir("NewDir")
4  print os.listdir(".")
6
```

```
['Documents', 'Music', 'Movies', 'Scripts']
['Documents', 'Music', 'Movies', 'NewDir', 'Scripts']
```

The os module

```
os.makedirs(path[, mode])
```

Recursive directory creation function. Like **mkdir()**, but makes all intermediate-level directories needed to contain the leaf directory. Raises an error exception if the leaf directory already exists or cannot be created.

```
1   import os
2   print os.getcwd()
3   os.makedirs("Scripts/Project1/testing")
4   print os.listdir("Scripts")
5   print os.listdir("Scripts/Project1")
7
```

```
'/home/bruno'
['Project1']
['testing']
```

The os module

```
os.remove(path)
```

Remove (delete) the file path. If path is a directory, an error is raised. For directories, use **rmdir()** instead.
Availability: Unix, Windows.

```
1   import os
2   print os.listdir("Scripts")
3   os.remove("Scripts/script2.py")
4   print os.listdir("Scripts")
5
7
```

```
['script1.py', 'script2.py']
['script1.py']
```

```
os.removedirs(path)
```

Remove (delete) the directory path. Only works when the directory is empty, otherwise, an error is raised.
Availability: Unix, Windows.

```
import os
print os.listdir(".")
os.remove("NewDir")
print os.listdir(".")
```

```
['Documents', 'Music', 'Movies', 'NewDir', 'Scripts']
['Documents', 'Music', 'Movies', 'Scripts']
```

```
os.removedirs(path)
```

Remove directories recursively. Works like rmdir() except that, if the leaf directory is successfully removed, removedirs() tries to successively remove every parent directory mentioned in path until an error is raised.

```
import os
os.remove("NewDir/SubDir")
```

The os module

```
os.rename(src, dst)
```

Rename the file or directory src to dst. If dst is a directory, an error will be raised.

Unix: if dst exists and is a file, it will be replaced silently if the user has permission.
Windows: if dst already exists, an error will be raised even if it is a file.

Availability: Unix, Windows.

The BioPython (Bio) module

The Biopython module, is a collection of tools and modules that have been developed

focusing on bioinformatic and computational biology problems. It has numerous functionalities such as:

- **Parsing** several bioinformatic file types (FastA, Clustalw, GenBank, PubMed, ExPASy, among others) into utilizable data structures that ease the processing of the data;
- Tools that easily perform common operations on sequences, such as **translation**, **transcription**, **reverse complement**.
- Interfaces (both local and remote) to common bioinformatic programms, such as **NCBI's BLAST** and **Clustalw** alignment program.
- **Downloading** files from public resources, such as NCBI databases

The BioPython (Bio) module

Whetting Your Appetite

- The **Seq** class adds a layer of information to the tradicional sequence strings through the inclusion of an alphabet that specifies the kind of sequence stores (ambiguous/unambiguous DNA, RNA and protein).

```
1  from Bio import Seq
2  from Bio.Alphabet import IUPAC
3  my_sequence =
   Seq.Seq("AGTGTCGATGTCGTGCTAGCTAGCTG",IUPAC.unambiguous_dna)
4  my_sequence
6
```

```
Seq('AGTGTCGATGTCGTGCTAGCTAGCTG', IUPACUnambiguousDNA())
```

- In most ways, *my_sequence* behaves like a regular sequence string and most basic string methods still apply

```
1  my_sequence.lower()
2  my_sequence[4:15]
3  my_sequence.count("G")
5
```

```
1| Seq('agtgtcgatgtcgtgctagctagctg', DNAAlphabet())
2| Seq('TCGATGTCGTG', IUPACUnambiguousDNA())
3| 4
```

The BioPython (Bio) module

Whetting Your Appetite

- However, it is now possible to easily perform basic sequence manipulations

```
my_sequence.reverse_complement()
RNA = my_sequence.transcribe()
RNA
Protein = RNA.translate(table="Standard")
Protein
```

```
1| Seq('CAGCTAGCTAGCACGACATCGACACT', IUPACUnambiguousDNA())
3| Seq('AGUGUCGAUGUCGUGCUAGCUAGCUG', IUPACUnambiguousRNA())
5| Seq('SVDVVLAS', IUPACProtein())
```

- And even to create mutable sequence strings

```
mutable_sequence = Seq.MutableSeq
("AGTGTCGATGTCGTGCTAG",IUPAC.unambiguous_dna)
mutable_sequence[:12] = "TTTTTTTTTTTT"
mutable_sequence
```

```
MutableSeq('TTTTTTTTTTTTGTGCTAG', IUPACUnambiguousDNA())
```

The BioPython (Bio) module

Whetting Your Appetite

The **SeqRecord** class allows identifiers and features to be associated with a sequence, creating sequence records much more richer in information:

- **seq**: The sequence itself
- **id**: A unique sequence identifier. Typically an accession number.
- **name**: A "common" name/id for the sequence as a string.
- **annotations**: A dictionary with additional information about the sequence

These features can be created manually, or imported directly from a database record

The BioPython (Bio) module

Whetting Your Appetite

- The **SeqRecord** class:

```
from Bio.Seq import Seq
my_sequence = Seq("ACGTAT")
from Bio.SeqRecord import SeqRecord
my_sequence_rec = SeqRecord(my_sequence)

my_sequence_rec.id = "TG123989"
my_sequence_rec.description = "My beloved sequence"
my_sequence_rec.name = "Made upinus"
my_sequence_rec.annotations["phred scores"] =
[20,30,20,50,10,23]
print my_sequence
```

```
ID: TG123989
Name: Made upinus
Description: My beloved sequence
Number of features: 0
/phred scores=[20, 30, 20, 50, 10, 23]
Seq('ACGTAT', Alphabet())
```

The BioPython (Bio) module

Whetting Your Appetite

- **SeqIO** is a module that provides a simple interface for working with assorted sequence file formats, but will only deal with sequences as SeqRecord objects. The most useful function is the Bio.SeqIO.parser() that can read sequence data as SeqRecord objects.

```
from Bio import SeqIO
for seq_record in SeqIO.parse("My_fasta.fas","fasta")
print seq_record
print seq_record.id
```

```
5     print seq_record.seq
6
7     for seq_record in SeqIO.parse("My_genbank.gb","genbank")
8     print seq_record
9     print seq_record.id
10    print seq_record.description
12
```

The BioPython (Bio) module

Whetting Your Appetite

- **SeqIO**:

The SeqIO.write() funtion can write a set of SeqRecord objects into a new file in a format specified by the user. You only need (i) one or more SeqRecord objects, a filename to write to, and a sequence format.

```
1     from Bio.Seq import Seq
2     from Bio.SeqRecord import SeqRecord
3     from Bio.Alphabet import generic_dna
4     Seq1 = SeqRecord(Seq("ACGTGA",generic_dna),id="first
5     sequence")
6     Seq2 = SeqRecord(Seq("CGTGTA",generic_dna),id="second
7     sequence")
8     Seq3 = SeqRecord(Seq("CTGTGA",generic_dna),id="third
9     sequence")
10    Records = [Seq1,Seq2,Seq3]
11
13    from Bio import SeqIO
      output_fasta = open("My_new_fasta.fas","w")
      SeqIO.write(Records,output_fasta,"fasta")
```