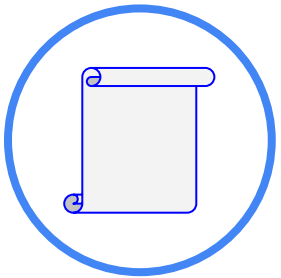
Python Basics II

20.11.19 / Us (again)

Today’s Focus

Part II

Anaconda



Virtual Environment

Challenges

Anaconda

- Python distribution which is popular for data analysis & scientific computing
- Open Source
- Available for MacOs, Windows & Linux
- Includes many popular packages like NumPy, SciPy, Matplotlib, etc.
- → **Jupyter Notebook:** Python development environment
- → **conda:** Platform independent package manager

Conda

- Powerful package & environment manager which is used with command line
  - *For any language → Python, R, Ruby, Lua, Scala, Java, JavaScript, C/ C++, FORTRAN*
- → installs, runs and updates packages and their dependencies
- → creates, saves, loads and switches between environments on your local computer
- → can be combined with continuous integration systems such as Travis CI and AppVeyor to provide testing of your code

Ok, let’s get started...

Google Drive link with files: https://tinyurl.com/rn6zxgs

Ok, let’s get started...


AIDS
WALK
NEW
YORK

Before we get started…
… one last thing ☝

Virtual Environments

- A tool in order to keep dependencies required by different projects separately by creating isolated python virtual environments
- This is one of the most important tools that most of the Python developers use.

...BUT WHY DO WE NEED THIS?!

The Problem

Scenario:

There is ***Project 1*** and ***Project 2***. Both projects have dependency on the same library, ***Project 3.*** What happens if both projects require **different versions** of **Project 3**. In other words, **Project 1** needs v1.0.1, but **Project 2** needs a newer version v.1.2.4.

The Solution!

- Python virtual environment creates for each project an isolated environment
- Each environment is able to depend on whatever version of the chosen library (independent of the other)
- Unlimited number of environments possible (since they are dictionaries containing a few scripts)
- Easily created with e.g. venv *(pip install & python3)* → then activate! →

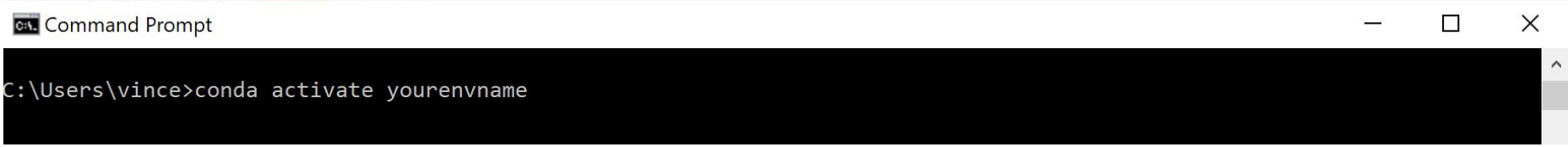
→ **But, we will do it a bit differently!**

More Info

Create Virtual Environment with Conda I

Create the virtual environment

Activate your environment



Install any package for your environment

Create Virtual Environment with Conda

Deactivate the virtual environment

Show all your environments

Remove a specific environment

Quick Recap!

Most Important Points from last time!

- Everything in Python are objects
  - There is a difference between mutable and immutable
- Code blocks are separated by indentation
- Comments with: # or ´´´…….´´´
- Data Types: Numbers (int, float, etc.), strings (“ ...”), lists ([...]), dictionaries ({keys: values})
- There are data type specific methods: .lower(), .append(), .sort(), etc.
- Monty Python

Ok, enough theory! 

→ Time for a ...

Challenge I

Control Structures

Conditionals

- True & False as Boolean objects of class = bool
- Bool are immutable
- Non zero value & non-null as True, otherwise False
- No “switch” statements like in other languages

Conditional statements in python

First of all, we look at the simple **if statement**

In [53]:
```
if 10 > 12:
    print("Change of history!")
```

Now, we look at the **if-else statement**

In [54]:
```
if 10 > 12:
    print("Change of history!") # watch out, you seperate code blocks just with the number of withespaces in python
else:
    print("Nothing new here!")
```

```
Nothing new here!
```

And lastly, we add a bit of **elif** here!

In [55]:
```
list_1 = [12,34, "Hanna", "Montana", []]

if 10 > 12: #False
    print("Change of history!")
elif 5 > 7: #False
    print("Still change of history!")
elif 12 in list_1: #True
    print("Ok, I got it!")
else:
    print("Nothing new here!")
```

```
Ok, I got it!
```

Conditionals

Ok, let's quickly check the **pass** statement

In [56]:
```
if 10 > 5:

print("Apfelstrudel")
```

```
  File "<ipython-input-56-670a2aeb198c>", line 3
    print("Apfelstrudel")
        ^
IndentationError: expected an indented block
```

But if we add a little **pass**

In [57]:
```
if 10 > 5:
    pass

print("Apfelstrudel")
```

```
Apfelstrudel
```

Ok, and last but not least --> Boolean operators

not: not True --> False and or

In [59]:
```
if 10 > 5 and 100 > 99:
    print("That was and!")

if 1 > 5 or 100 > 99:
    print("That was or!")

if not False:
    print("That is obviously True")
```

```
That was and!
That was or!
That is obviously True
```

Loops

- “For” loops
- “While” loops
- Control statements:
  - break
  - continue
  - pass

While

In [65]:
```
number = 0

while number < 5 and True:
    print(number)
    number += 1
```

```
0
1
2
3
4
```

For

Example 1!

In [61]:
```
for letter in "deoxyribonucleic acid":
    print("That's hell of a word!")
```

```
That's hell of a word!
That's hell of a word!
That's hell of a word!
That's hell of a word!
That's hell of a word!
That's hell of a word!
That's hell of a word!
That's hell of a word!
That's hell of a word!
That's hell of a word!
That's hell of a word!
That's hell of a word!
That's hell of a word!
```

Loops I

__Example 2!__

In [63]:
```
random_list = ["Monkey", "Chips", "Games of Thrones", "Surf Gloves"]
for stuff in random_list:
    print(stuff)
```

```
Monkey
Chips
Games of Thrones
Surf Gloves
```

__Example 3!__

In [77]:
```
random_list = ["Monkey", "Chips", "Games of Thrones", "Surf Gloves"]

print(range(len(random_list))) # will create a range of numbers according to length of list

for index in range(len(random_list)):
    print(random_list[index])
```

```
range(0, 4)
Monkey
Chips
Games of Thrones
Surf Gloves
```

Loops II

First: __break!__

In [80]:
```
number = 0

while number < 5:
    if number == 3:
        print("That is not my favourite number")
        break
    else:
        print(number)
    number += 1
```

```
0
1
2
That is not my favourite number
```

Now: __continue!__

In [88]:
```
for letter in "Apple":
    if letter == "A":
        print("That's cool")
        continue
    else:
        print(letter)
```

```
That's cool
p
p
l
e
```

Once again: __pass!__

In [89]:
```
for letter in "Pear":
    if letter == "H":
        pass
    else:
        print(letter)
```

```
P
e
a
r
```

Functions

- Functions argument:
  - Required arguments
  - Keyword arguments
  - Default arguments
- *args and **kwargs

Functions in python

Syntax

In [91]:
```
def sum_calc(a,b):
    c = a + b
    print(c)

sum_calc(8,7)
```

```
15
```

Functions

Keyword arguments

function calls can be identified by parameter name!

In [97]:
```
def greeting(greeting,name):
    phrase = greeting + ' ' +name + "!"
    print(phrase)

greeting(name='James', greeting='Hello')
```

```
Hello James!
```

Required arguments

arguments have to be in the right order!

In [96]:
```
def greeting(greeting,name):
    phrase = greeting + ' ' +name + "!"
    print(phrase)

greeting('Hello','James')
```

```
Hello James!
```

Default arguments

function has a default value in the function declaration, when value is not provided in call

In [101]:
```
def call_people(name, age = 35):
    string = name + " is " + str(age) + " old!"
    print(string)

call_people(name="Wolfgang")
```

```
Wolfgang is 35 old!
```

Functions

Variable-lengths arguments

used when there is an unspecified amount of arguments

args : By using * as syntax you can take in a variable amount of arguments than the number of formal arguments you previously defined - by convention with word **args**.

In [104]:
```
def randomFun(arg1, *argv):
    print(arg1)
    for arg in argv:
        print(arg)

randomFun("First argument", "Ulrich", 2, [1,2,3])
```

```
First argument
Ulrich
2
[1, 2, 3]
```

kwargs : Now you can pass a keyworded, variable-length argument list. **Remember:** A keyword argument is where you have to name the variable as you call it with the function

kwargs is like being a dictionary that maps each keyword to the value that we pass. --> this means that when we iterate over the kwargs, there doesn't have to be an order

In [113]:
```
def randomFun(**kwargs):
    for key, value in kwargs.items():
        print("{} + {}".format(key,value))

randomFun(first ='Ok', mid ='I get', last='it')
```

```
first + Ok
mid + I get
last + it
```

Modules

- consists of python code that defines classes, functions & variables
- Organize code by grouping related code
- Import statement
- → import … from

Modules in python

__let's have a loot at a sample module --> random

There are several ways in order to import the methods from the random module but, every python file (ends with .py) can be imported as a module in your script, this is a great way to structure program

Option 1

```
In [127]: dir(random)
```

```
Out[127]: ['BPF',
           'LOG4',
           'NV_MAGICCONST',
           'RECIP_BPF',
           'Random',
           'SG_MAGICCONST',
           'SystemRandom',
           'TWOPI',
```

```
In [126]: help(random.randint)
```

```
Help on method randint in module random:

randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.
```

```
In [138]: import random

          def get_num(arg1):
              rand_num = random.randint(0,5)
              sum_tot = rand_num + arg1
              print("The sum is: " + str(sum_tot))

          get_num(5)
```

```
The sum is: 5
```

Modules

Option 2

In [139]:
```
from random import randint, sample # will only import randint and sample from random

def get_num(arg1):
    rand_num = randint(0,5) # there is no need of random. now
    sum_tot = rand_num + arg1
    print("The sum is: " + str(sum_tot))

get_num(5)
```

```
The sum is: 6
```

Option 3

In [141]:
```
from random import randint as ri

def get_num(arg1):
    rand_num = ri(0,5) # there is no need of random. now (now with ri)
    sum_tot = rand_num + arg1
    print("The sum is: " + str(sum_tot))

get_num(5)
```

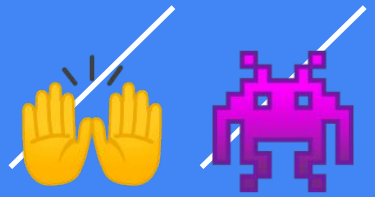
```
The sum is: 10
```

one last thing to mention!

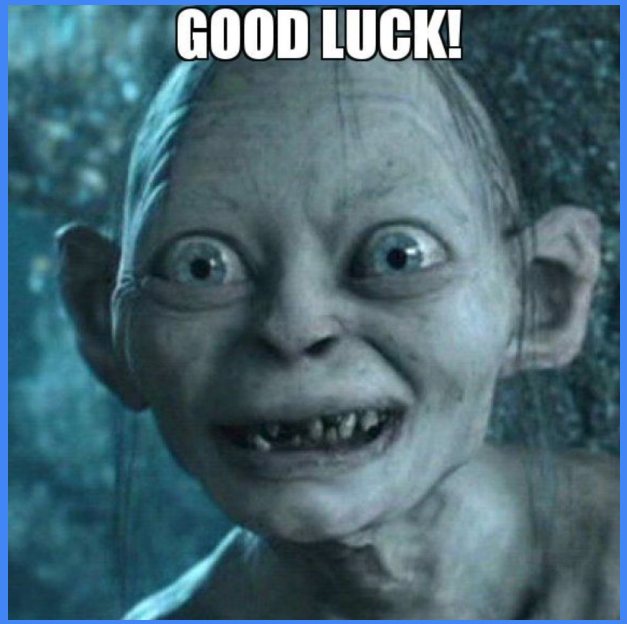
In [ ]:
```
from random import * # will import everything --> not really the best way to do it
```

Okay guys, wake up! 

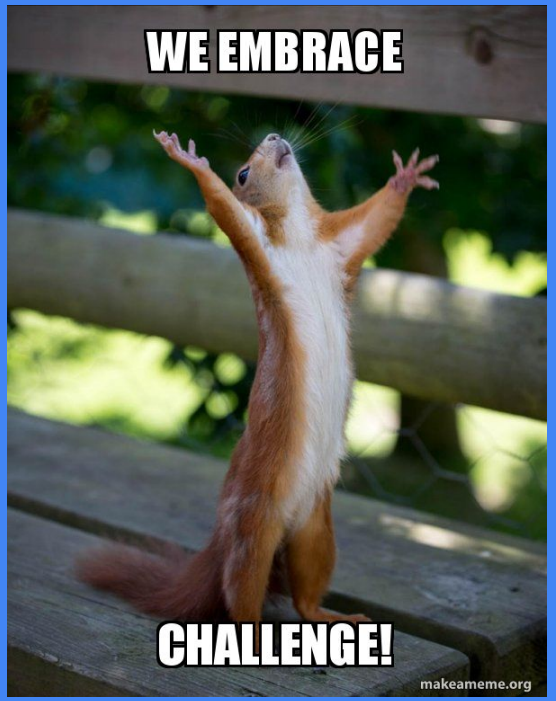
It is your turn, now!

Let’s code!!! 

Challenge II

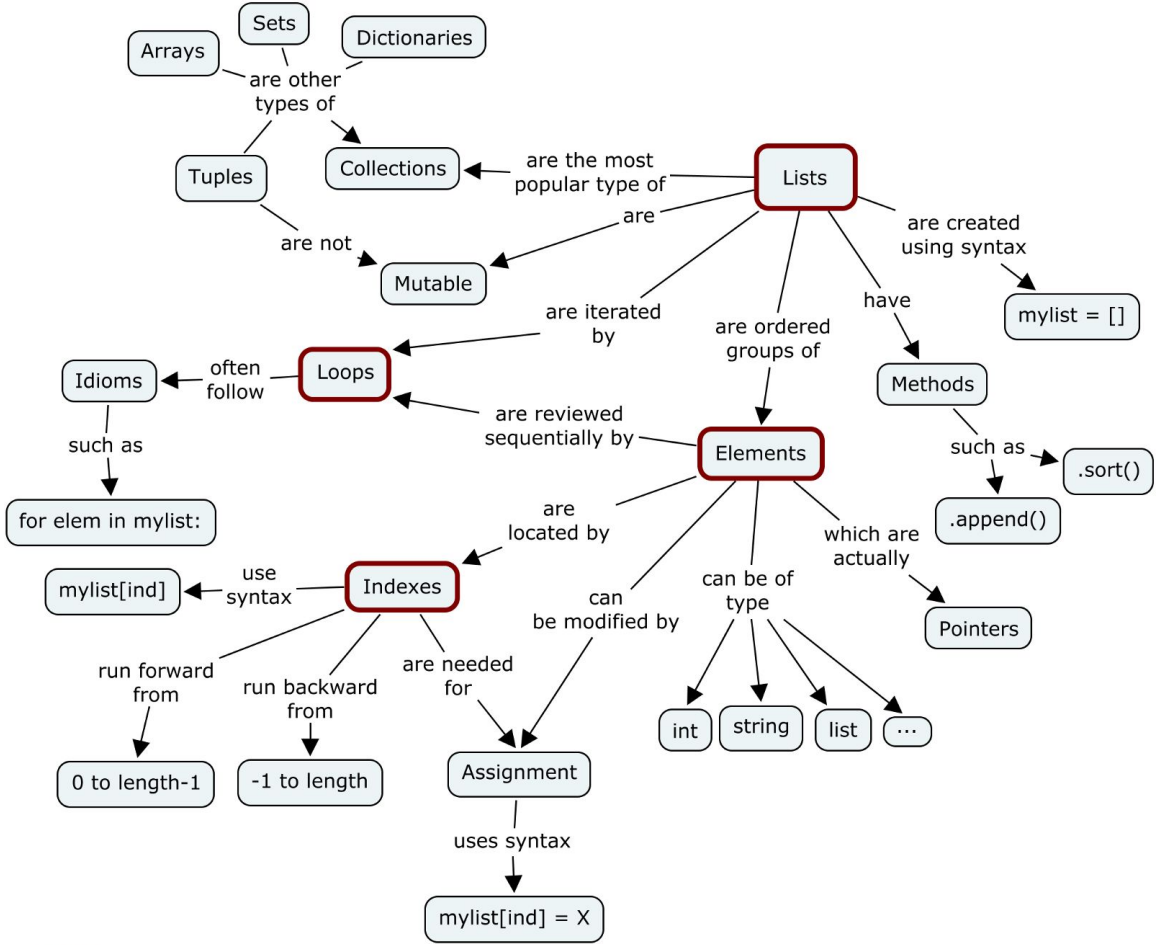


Final Challenge II



Need some inspiration…?

Summary


Sets
Arrays
Dictionaries
are other types of
Tuples
Collections
are the most popular type of
Lists
are
are not
Mutable
are created using syntax
mylist = []
are iterated by
are ordered groups of
have
Idioms
often follow
Loops
Methods
are reviewed sequentially by
Elements
such as
for elem in mylist:
such as
.sort()
.append()
are located by
which are actually
Pointers
mylist[ind]
use syntax
Indexes
can be modified by
can be of type
run forward from
run backward from
are needed for
int
string
list
...
0 to length-1
-1 to length
Assignment
uses syntax
mylist[ind] = X

Thanks!

Python team

Wiki:
https://wiki.tum.de/display/ldv/Info

Mail:
pythonworkshop.tum@gmail.com

Web:
https://www.ei.tum.de/startseite/

Git:
https://gitlab.ldv.ei.tum.de/daedalus/python