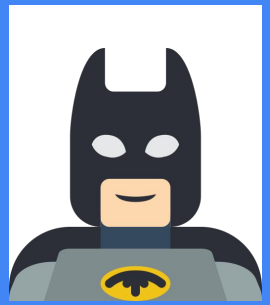
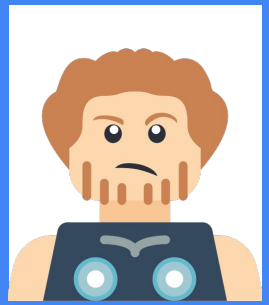
Python Basics I

13.11.19 / Us

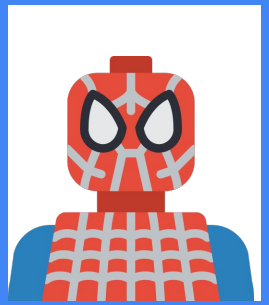
Team



Hamid



Xavi



Vincent

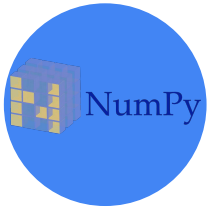
What is the plan...

Basics I

06.11.19
AND
13.11.19

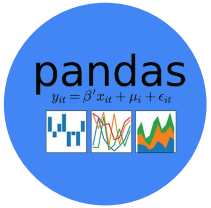
Basics II

20.11.19



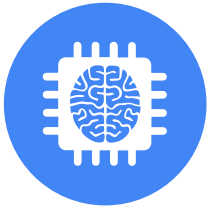
NumPy

27.11.19



Pandas

04.12.19



Machine Learning

11.12.19

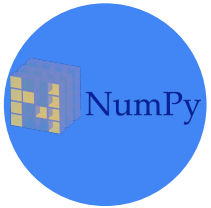
GitLab: https://tinyurl.com/y5y7oj6n

Basics I

06.11.19
AND
13.11.19

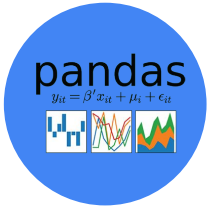
Basics II

20.11.19



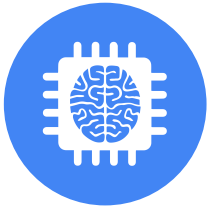
NumPy

27.11.19



Pandas

04.12.19



Machine Learning

11.12.19

What is Python?!

- General Purpose Language
- Object oriented (e.g. inheritance)
- Open Source and Multi-Platform
- Libraries, Third Party Utilities, Built-in Types and Tools
- *Monty Python

Installing Python

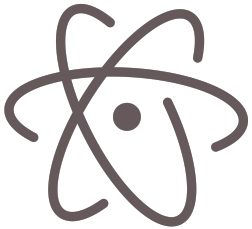
- Python is default pre-installed in most of the UNIX based systems → Linux & MacOs
- **Download Link**: https://www.python.org/downloads/

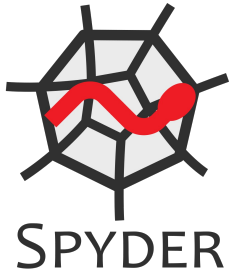
- **Download Link**: https://www.python.org/downloads/
  - Latest version is Python 3.7.3
- Edit Environment Variable to run python scripts from the terminal // **or accept in Download process!**

We are gonna do it a bit differently...

- **Why Anaconda!**
  - Easy python install
  - 720 open source packages (e.g. NumPy, Matlibplot, etc.)
  - Conda package manager → easy use of virtual environment
- **Why Jupyter Notebook**
  - Cool environment where you can run your code, look at the results & easily modify
  - Easy sharing & virtual environment
  - Nice take-away!

But what about other stuff?





- **Code Editors with Python support**
  - Text editor with syntax highlighting
  - Visual Studio Code, Atom Editor
- **Python-Specific IDEs**
  - Program dedicated to software development (code editor + execution/debugging tools)
  - PyCharm, Spyder

Syntax

- **Indentation:**
  - Structures the blocks of your code - number of spaces is variable, but code within the same block must be indented the same amount - best practice: 4 spaces
- **The header line:**
  - For compound statements such as: while, if, def, etc. must be terminated with a colon (:)
- **Printing on the screen**
  - print(“Hello World”)
- **Reading the keyboard**
  - Fav_Number = input(“Tell me your number? ”)
- **Comments:**
  - Single line → # This is just a random comment
  - Multiple lines →```Oh my good, this is awesome```
- **Python files**
  - End with .py

Variables

- **Dynamically typed:**
  - You don’t have to declare the type of your variable
- **Automatically change:**
  - You can easily change the data type of your variable by just assigning it newly
- **Single value for all:**
  - In Python you can assign a single value to multiple variables
- **Multiple objects to multiple variables:**
  - Further, you can assign multiple objects to multiple variables
- **Naming convention:**
  - Lowercase single letter, separate words with underscores (_) to improve readability
  - For Class → CamelCase

Data Types

Numbers

- Immutable in python
  → cannot change their values
- 3 built-in data types:
  - Integers
  - Floating point numbers
  - Complex numbers

Functions	Explanation
int(x)	Convert x in integer
float(x)	Convert x in float
abs(x)	Absolute value of x
exp(x)	Exponential of x
log(x)	Natural logarithm of x
pow(x, y)	Value of x**y
sqrt(x)	Square root of x

Math Operations in Python

+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
*** Multiplication**	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
**** Exponent**	Performs exponential calculation on operators	a**b = 10 to the power 20
//	Floor Division	9 // 2 = 4

Strings

- Immutable in python
- Update a string by assign a variable to another string
- No <<char>> type
- Single, double or triple quotes for strings
- Index starts from 0 and -1 from the end along

Functions & Methods	Explanation
str(x)	Convert x in string
len(string)	Total length of string
string.count(“am”)	Counts ”am” in string
string.lower()	Converts string to lowercase
string.upper()	Converts string to uppercase
string.replace(old, new)	Replaces all occurrences in string with new string
string.split(separator, max)	Splits string with defined separator and gives list
string.strip()	Removes all leading and trailing whitespaces in string

String Operators I

String Operators

Concatenation

Adds the value on both sides of the operator

In [2]:
```
string_1 = 'Hello'
string_2 = 'Python Master!'
string_sum = string_1 + string_2
print(string_sum)
```

```
HelloPython Master!
```

Repetition

In [3]:
```
string_1 = 'movies'
string_2 = string_1*2
print(string_2)
```

```
moviesmovies
```

String Operators II

List Slicing

Slices through the string like a list. Range slice gives you the characters from a given range, where just the index gives you the referring character

In [10]:
```
string_1 = "Apfelkuchen"
index_1 = string_1[0]
index_2 = string_1[-1]
range_1 = string_1[1:4]
range_2 = string_1[:5]
print('Hier ist der Index 1: ' + index_1)
print('Hier ist der Index 2: ' + index_2)
print('Hier ist die Range 1: ' + range_1)
print('Hier ist die Range 2: ' + range_2)
```

```
Hier ist der Index 1: A
Hier ist der Index 2: n
Hier ist die Range 1: pfe
Hier ist die Range 2: Apfel
```

Membership with in ¶

Checks if the character exists in the given and return True, if it is the case...

In [9]:
```
string_1 = "Allerlei Sachen"
condition_1 = "All" in string_1
print(condition_1)
```

```
True
```

String Formatting in Python

String Formatting

There are two ways in order to format your strings. The difference comes with the updates version of python (Python 3), where f-strings are introduced for the first time. However, with the old version, you cannot fail.

Old School String Formatting

In [1]:
```
num = 10
string_1 = "I have {} favourite movies".format(num)
print(string_1)
```

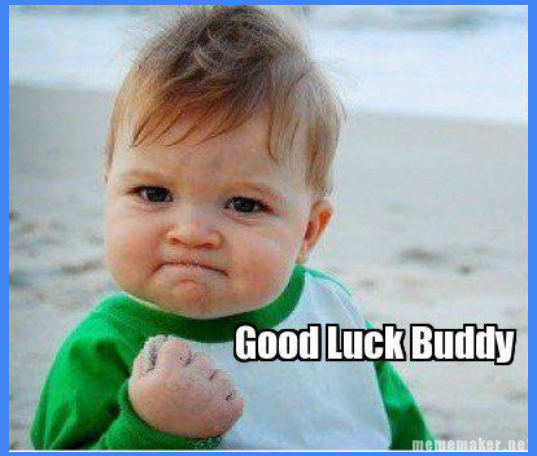
```
I have 10 favourite movies
```

F-String Python 3

In [2]:
```
num = 10
string_1 = 'movies'
string_2 = f"I have {num} favourite {string_1}"
print(string_2)
```

```
I have 10 favourite movies
```

Challenge 1



Lists

- Mutable in python
- Ordered group of elements - don't have to be same data type
- Most string operations work for lists
- Can have sublists, and sublists of sublists

Functions & Methods	Explanation
len(list)	Total length of list
cmp(list1,list2)	Compares elements of both lists
max(list)	Return item with highest value
min(list)	Return item with lowest value
list.append()	Appends element to list
list.insert(index, element)	Inserts element to list according to index
list.remove(element)	Removes element from list
list.sort()	Sorts list in place
list.reverse()	Reverses elements in list in place

List Comprehension

List Comprehension

Each list comprehension contains an **expression** and then a **for** clause

In [11]:
```
list_1 = [1,2,3,4,5]
list_2 = [x+1 for x in list_1]
print(list_2)
```

```
[2, 3, 4, 5, 6]
```

In [12]:
```
list_1 = [1,2,3,4,5]
list_2 = [x+1 for x in list_1]
list_3 = [z*2 for z in [x+1 for x in list_1]]
print(list_3)
```

```
[4, 6, 8, 10, 12]
```

In [ ]:
```

```

List Operations I

List Operations

Just a quick summery of most common list operations

Append, Insert & Remove

!Wow, the same list is showed twice, why?!

In [3]:
```
list_1 = [90,12,3,["Katze",2], "Hund"]
list_2 = list_1
list_1.append(99)
list_2.insert(2,'Element')

print(list_1)
print(list_2)
```

```
[90, 12, 'Element', 3, ['Katze', 2], 'Hund', 99]
[90, 12, 'Element', 3, ['Katze', 2], 'Hund', 99]
```

Now, we will make some small changes!

In [4]:
```
list_1 = [90,12,3,["Katze",2], "Hund"]
list_2 = list_1.copy()
list_1.append(99)
list_2.insert(2,'Element')

print(list_1)
print(list_2)
```

```
[90, 12, 3, ['Katze', 2], 'Hund', 99]
[90, 12, 'Element', 3, ['Katze', 2], 'Hund']
```

In [5]:
```
list_1 = [90,12,3,["Katze",2], "Hund"]
list_1.remove(90)
print(list_1)
```

```
[12, 3, ['Katze', 2], 'Hund']
```

List Operations II

List Operations

Just a quick summery of most common list operations

Sort & Reverse

first let us look at sort

In [18]:
```
list_1 = [90,12,20,23]
list_2 = ['Manfred', 'Stefan','Lisa','Hanna',]

list_1.sort()
list_2.sort()
print(list_1) #from bottom to top
print(list_2)

list_1.sort(reverse = True) #from top to bottom
print(list_1)

list_2 = [90,12,20,23]
```

```
[12, 20, 23, 90]
['Hanna', 'Lisa', 'Manfred', 'Stefan']
[90, 23, 20, 12]
```

now reverse

In [19]:
```
list_1 = [90,12,20,23]
list_2 = ['Manfred', 'Stefan','Lisa','Hanna',]

list_1.reverse()
list_2.reverse()
print(list_1) #from bottom to top
print(list_2)
```

```
[23, 20, 12, 90]
['Hanna', 'Lisa', 'Stefan', 'Manfred']
```

In [ ]:
```

```

Tuples

- Immutable in python
- Faster than lists and protect unwanted change
- Same rules like with lists regarding functions, indexing & methods
- Single value tuple still comma need

Tuples in python

In [32]:
```
tuple_1 = ("tuples", "are", "awesome")
tuple_2 = ("still", "great")
single_tuple = (1,) # we need the comma here
print(tuple_1)
print(tuple_2)
print(single_tuple)
```

```
('tuples', 'are', 'awesome')
('still', 'great')
(1,)
```

what happens if we try to change a value

In [26]:
```
tuple_1 = ("tuples", "are", "awesome")
tuple_2 = ("still", "great")
tuple_1[1] = "were"
print(tuple_1)
print(tuple_2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-26-35804d59ebb5> in <module>
      1 tuple_1 = ("tuples", "are", "awesome")
      2 tuple_2 = ("still", "great")
----> 3 tuple_1[1] = "were"
      4 print(tuple_1)
      5 print(tuple_2)

TypeError: 'tuple' object does not support item assignment
```

Tuples

ok, and what about this...

In [27]:
```
tuple_1 = ("tuples", "are", "awesome")
tuple_2 = ("still", "great")
tuple_2 = tuple_1
print(tuple_1)
print(tuple_2)
```

```
('tuples', 'are', 'awesome')
('tuples', 'are', 'awesome')
```

so, tuples are immutable, but we still can extend the tuple content with a trick...

In [28]:
```
tuple_1 = ("tuples", "are", "awesome")
tuple_2 = ("still", "great")
tuple_2 = tuple_1 + tuple_2
print(tuple_1)
print(tuple_2)
```

```
('tuples', 'are', 'awesome')
('tuples', 'are', 'awesome', 'still', 'great')
```

Dictionaries

- Like unordered Hash-table with keys & values
- Keys must be immutable
- Thus, dictionaries are mutable
- Access values with kex indexing

Functions & Methods	Explanation
dict1.keys()	Returns list of keys
dict1.values()	Returns list of values
dict1.items()	Returns list of key, value tuple pairs
dict1.get(key, default=None)	Returns value of key, if there is none, then default
Key in dict_1	Returns True if key exists, False otherwise
dict1.update(dict2)	Adds dict2 keys & values to dict1
dict1.clear()	Removes all elements from dict
len(dict1)	Total number of keys, values

Dictionary Operations

Dictionaries in python

In [44]:
```
dict_1 = {"item 1": 5, "item 2": 3, "item 3": 10}
dict_2 = {1: "Tomas", 2: "Sandra", 3: "Luca"}

print(dict_1)
print(dict_1.keys())
print(dict_2.values())
```

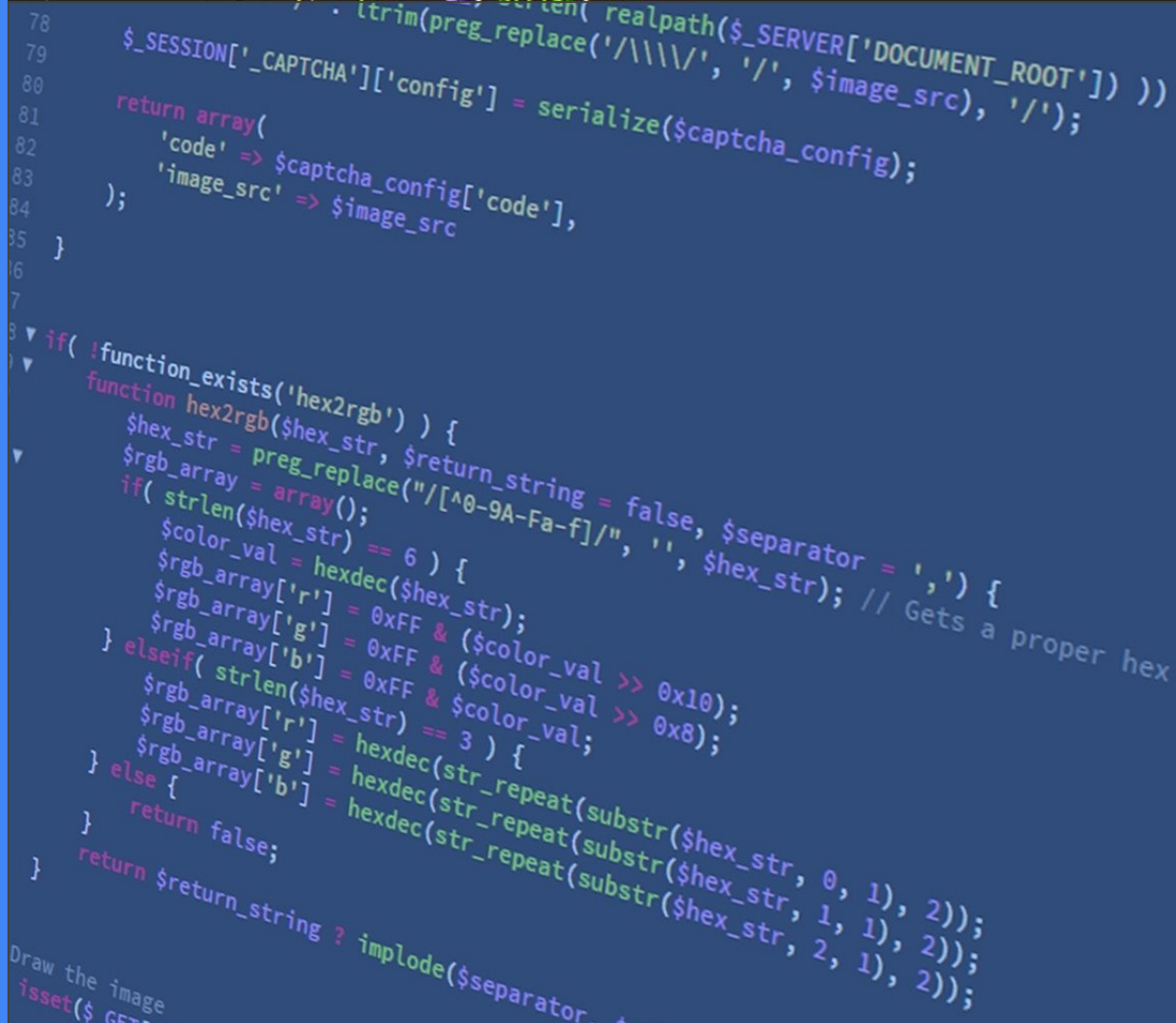
```
{'item 1': 5, 'item 2': 3, 'item 3': 10}
dict_keys(['item 1', 'item 2', 'item 3'])
dict_values(['Tomas', 'Sandra', 'Luca'])
```

what about the other methods

In [52]:
```
dict_1 = {"item 1": 5, "item 2": 3, "item 3": 10}
dict_2 = {1: "Tomas", 2: "Sandra", 3: "Luca"}
dict_3 = {"Apple": 2.5, "Melon": 1.5, "Bag": 0.5,}
condition = dict_1.get("Marvel", None)
a = len(dict_1)
b = 1 in dict_1
dict_1.update(dict_3)

print(condition)
print(a)
print(b)
print(dict_1)
```

```
None
3
False
{'item 1': 5, 'item 2': 3, 'item 3': 10, 'Apple': 2.5, 'Melon': 1.5, 'Bag': 0.5}
```

Ok, ok,...
there is one thing which we have to mention...



Mutable vs. Immutable

Difference between mutable and immutable

Everything in Python is an object, so every variable holds an object instance. When an object is initiated, it is assigned a unique object id (it is like the memory address). Its type is defined at runtime and once set can never change, however its state can be changed if it is mutable. Simple put, a mutable object can be changed after it is created, and an immutable object can't.

In [8]:
```
x = 10
y = 10

print(id(x))
print(id(y))
```

```
1666111808
1666111808
```

In [11]:
```
x = x + 1

print(id(x))
print(id(y))
print("So there is a change of address, I did not change the original object!")
```

```
1666111856
1666111808
So there is a change of address
```

In [26]:
```
m = [1,2,3,4,5,6,7,8]
n = m
print(id(m))
print(id(n))
```

```
111869544
111869544
```

In [29]:
```
print(m.pop())
print(id(m))
print(id(n))
print("I did  change the original object!")
print(m)
```

```
6
111869544
111869544
I did  change the original object!
[1, 2, 3, 4, 5]
```

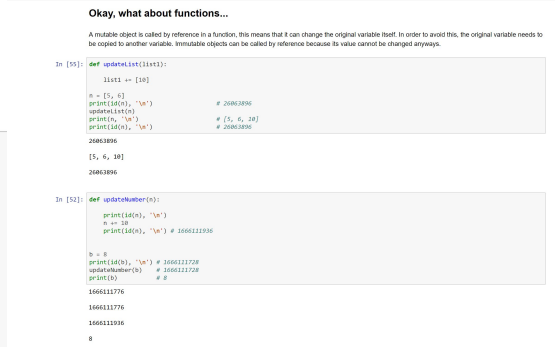
Mutable vs. Immutable II

Okay, there is another thing to remember...

We know that tuples are immutable. That means value of a tuple can't be changed after it is created. But the "value" of a tuple is infact a sequence of names with unchangeable bindings to objects. The key thing to note is that the bindings are unchangeable, not the objects they are bound to.

In [44]:
```
t1 = ('holberton', [1, 2, 3])
print(t1)
print(id(t1), '\n')

t2 = ('amazing', 'spiderman')
print(t2)
print(id(t2), '\n')

t1 = t1 + t2
print(t1)
print(id(t1), '\n') # So there are three different objects now

# But what if I try to change the value of the list
t1[1].append('A New World!')

print(t1)
print(id(t1), '\n')

# But what if I try to change the value of the holberton
t1[0] = ('Wubba lubba dub dub')

print(t1)
print(id(t1), '\n')
```

```
('holberton', [1, 2, 3])
95508240

('amazing', 'spiderman')
95514136

('holberton', [1, 2, 3], 'amazing', 'spiderman')
117088096

('holberton', [1, 2, 3, 'A New World!'], 'amazing', 'spiderman')
117088096
```




```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-44-5af3c04fd624> in <module>
     18
     19 # But what if I try to change the value of the holberton
---> 20 t1[0] = ('Wubba lubba dub dub')
     21
     22 print(t1)

TypeError: 'tuple' object does not support item assignment
```

Mutable vs. Immutable III

Okay, what about functions...

A mutable object is called by reference in a function, this means that it can change the original variable itself. In order to avoid this, the original variable needs to be copied to another variable. Immutable objects can be called by reference because its value cannot be changed anyways.

In [55]:
```
def updateList(list1):

    list1 += [10]

n = [5, 6]
print(id(n), '\n')                  # 26063896
updateList(n)
print(n, '\n')                      # [5, 6, 10]
print(id(n), '\n')                  # 26063896
```

```
26063896

[5, 6, 10]

26063896
```

In [52]:
```
def updateNumber(n):

    print(id(n), '\n')
    n += 10
    print(id(n), '\n') # 1666111936


b = 8
print(id(b), '\n') # 1666111728
updateNumber(b)    # 1666111728
print(b)           # 8
```

```
1666111776

1666111776

1666111936

8
```

Challenge 2

One final thing...*always*

look on the bride side of life

Thanks!

Python Team

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

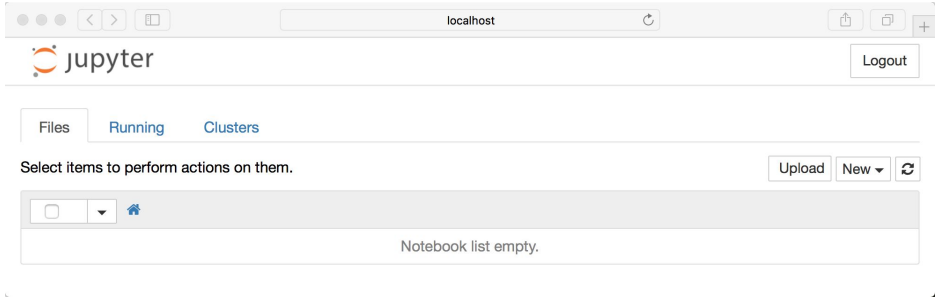
Mail:
pythonworkshop.tum@gmail.com

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

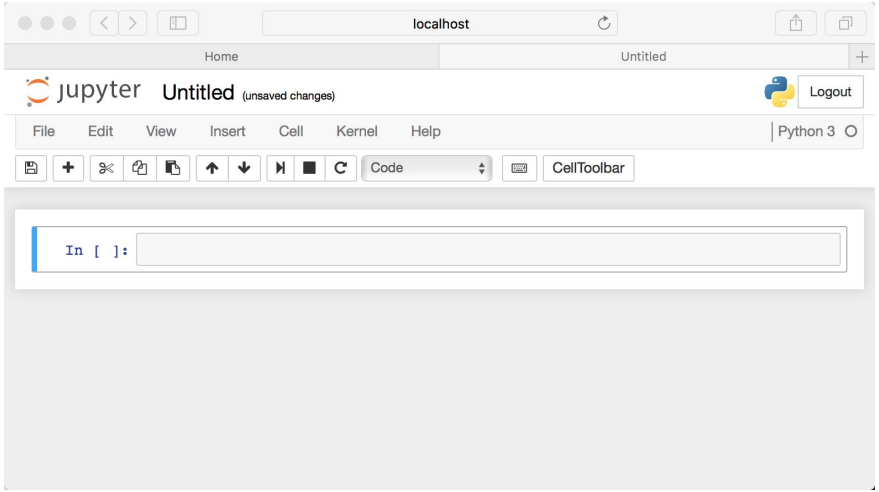
Git:
https://gitlab.ldv.ei.tum.de/daedalus/python

Backup - Jupyter Notebook - Overview

Main Menu:

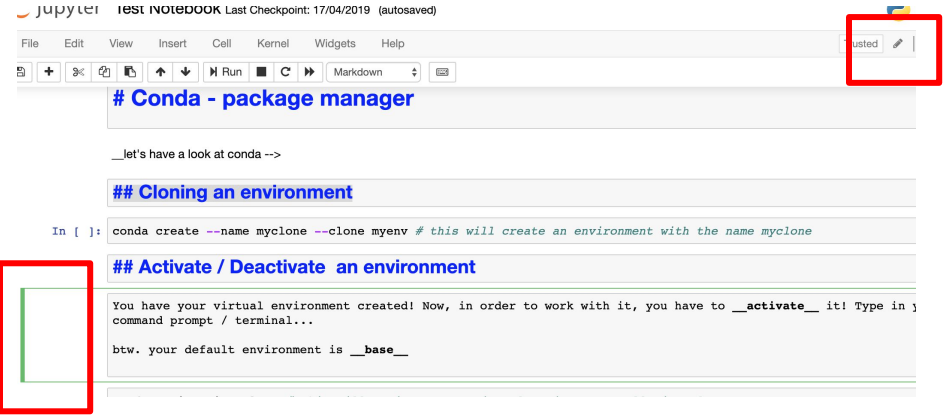


Creating a Notebook

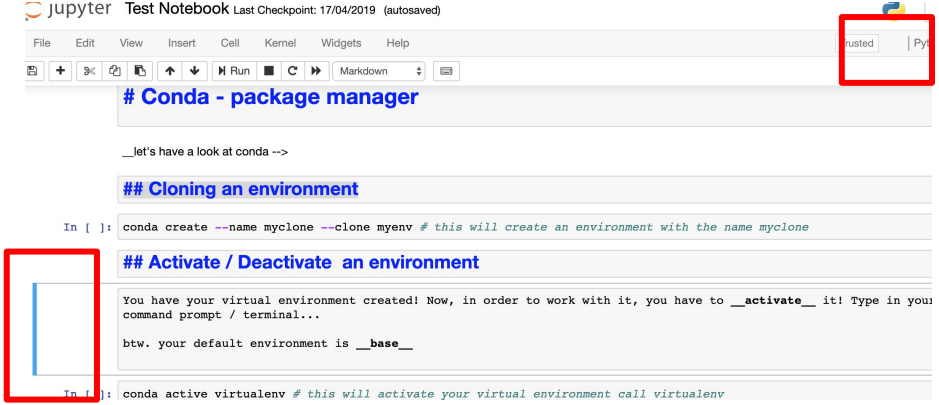


Backup - Jupyter Notebook - Overview

Edit Mode:

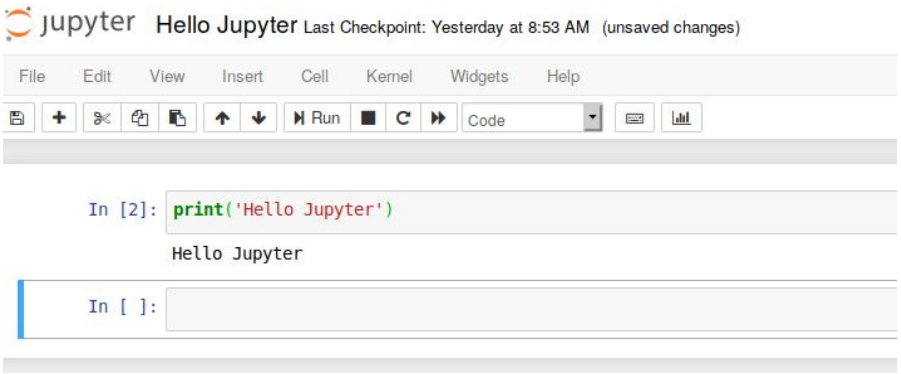


Command Mode:

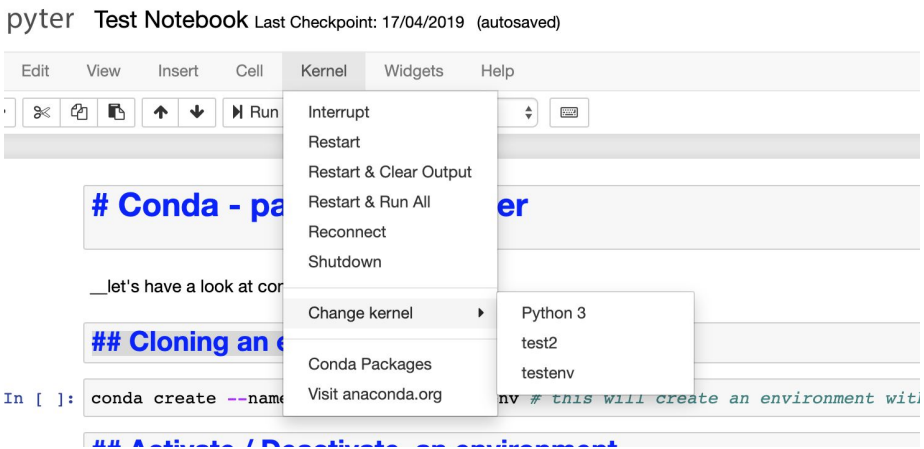


Backup - Jupyter Notebook - Overview

Run a cell:

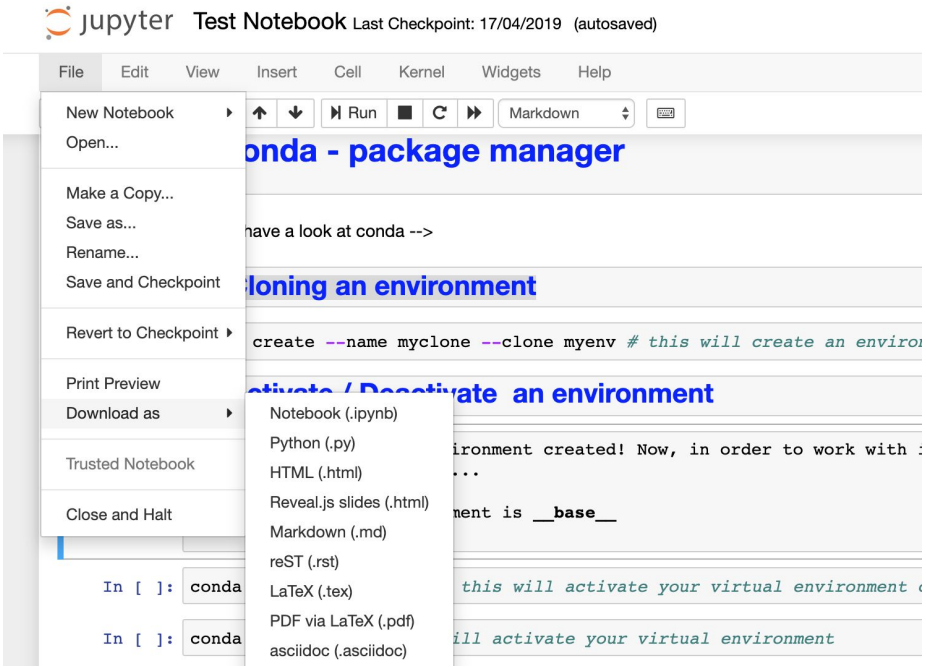


Change kernel:



Backup - Jupyter Notebook - Overview

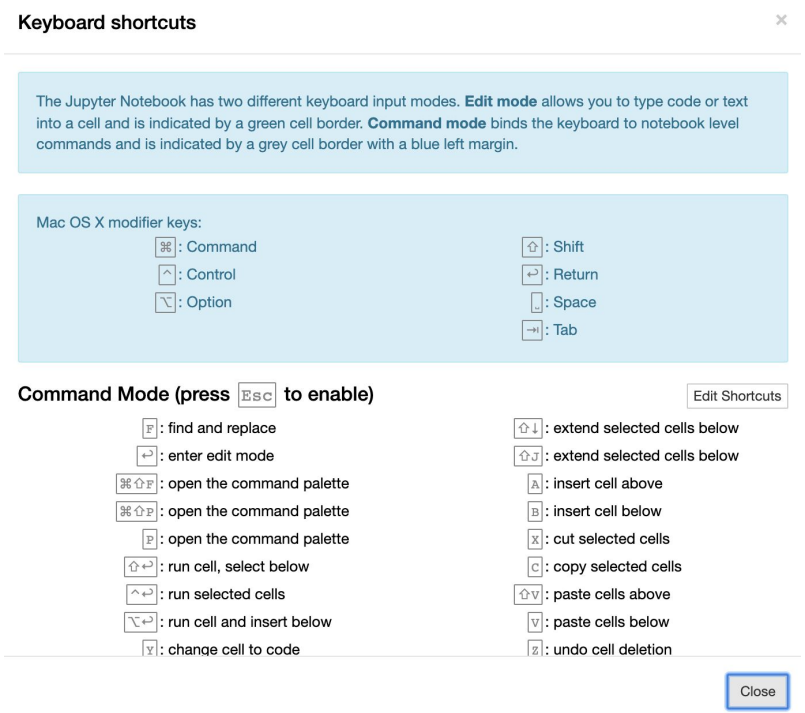
Export a Notebook:



Backup - Jupyter Notebook - Overview

Just press:
“H”
on your keyboard

Shortcuts:



Keyboard shortcuts

The Jupyter Notebook has two different keyboard input modes. **Edit mode** allows you to type code or text into a cell and is indicated by a green cell border. **Command mode** binds the keyboard to notebook level commands and is indicated by a grey cell border with a blue left margin.

Mac OS X modifier keys:

- ⌘: Command
- ^: Control
- ⌥: Option
- ⇧: Shift
- ↵: Return
- ␣: Space
- ⇥: Tab

Command Mode (press Esc to enable) — Edit Shortcuts

- F: find and replace
- ↵: enter edit mode
- ⌘⇧F: open the command palette
- ⌘⇧P: open the command palette
- P: open the command palette
- ⇧↵: run cell, select below
- ^↵: run selected cells
- ⌥↵: run cell and insert below
- Y: change cell to code
- ⇧↓: extend selected cells below
- ⇧J: extend selected cells below
- A: insert cell above
- B: insert cell below
- X: cut selected cells
- C: copy selected cells
- ⇧V: paste cells above
- V: paste cells below
- Z: undo cell deletion

Close