Python Intermediate

Kaju Bubanja

bubanja.kaju@gmail.com

It's nice to have you here today

About You

- Your major / occupation
- Your programming experience
- Your goals for this course

About me

- Bachelor in CS
- Master in Neural Systems and Computation
- Work at ces ag which make the speed traps in Zürich ;)
- I mainly program with Python and C++
- I used to build model rockets :)

Please Feel Free to Always Ask Questions

- Questions are a natural part of the learning process and you're always allowed to ask them
- **Asking questions is an integral part of this course**
- Even if you have a feeling that you're question might "not be good enough," or you don't understand a concept "even if it should be easy to do so," please ask the question nonetheless
  - For one, it gives me the possibility to try and come up with better / clearer explanations

Learning By Doing (and Making Errors)

- **Programming is best learned by doing**
- Don't be afraid to try stuff out in Python and make errors
  - Errors are a vital part of the learning process and help you understand situations much better

- If you should get stuck on an error during a programming exercise, please always feel free to call for my help or the help of fellow students
- Also, don't be afraid to use pen and paper to solve the exercises or when you are trying to understand a specific concept
  - For one, it helps a lot to step away from the computer from time to time
  - It also helps a lot to write down the immediate steps when trying to understand a complicated concept

Feedback

- I'm very thankful for all the feedback I get (be it positive or negative), since I want you to feel comfortable and I love to improve my courses and my teaching skills
  - Course is moving too fast?
  - I'm not speaking clearly enough?
  - Please feel free to inform me about anything whenever you feel like it ☺

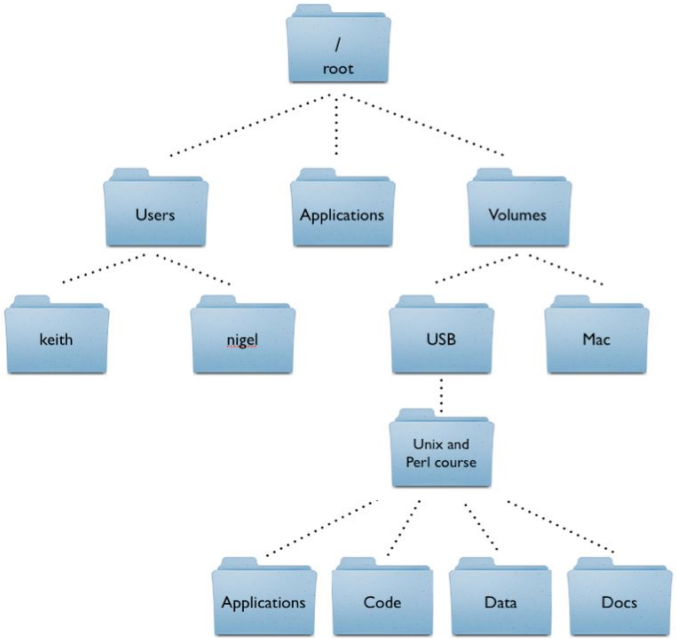
Learning Objectives for This Course

- Computer basics
- Learning to use the terminal
- Best practices for structuring your Python project
- Environments and Reproducibility
- Installing and Importing of Packages
- Functions with multiple arguments
- Exceptions and Debugging
- Basics of Object-oriented programming in Python
- A deeper dive into File I/O
- Handling web resources
- Basics of databases
- Capstone exercise (combining most of the topics)

Computers

File Tree

- Each computer has a file tree
- / is the root on unix systems
- C: is the root on windows systems
- `~` is the home folder, a shortcut for
- /home/<your_username> on Linux
- /users/<your_username> on Mac
- c:/users/<your_username> on windows



The Current Working Directory

- Every program that runs on your computer has a *current working directory*
  - It's the directory from where the program is executed / run
  - *Folder* is the more modern name for a directory
- The *root directory* is the top-most directory and is addressed by `/`
  - A directory `mydir1` in the root directory can be addressed by `/mydir1`
  - A directory `mydir2` within the `mydir1` directory can be address by `/mydir/mydir2`, and so on

Absolute and Relative Paths

- An *absolute path* begins always with the root folder, e.g. `/my/path/...`
- A *relative path* is always relative to the program's current working directory
  - If a program's current working directory is `/myprogram` and the directory contains a folder `files` with a file `test.txt`, then the relative path to that file is just `files/test.txt`
  - The absolute path to `test.txt` would be `/myprogram/files/test.txt` (note the root folder `/`)

Terminals


When I open terminal in front of non IT people
Will you please listen? I'm not a hacker.


*Non IT people
He is the hacker!
made with mematic

The terminal

- The terminal is a way to run commands on your computer
- Below is a list of basic commands
- `cd venv` Change Directory to the venv folder, without argument it brings you to your home folder
- `pwd` Prints your current Working Directory
- If you start typing a command and press double tab you can see possible autocompletions, single tab autocompletes an already started input

  `cd Do` in the home folder will show you `Downloads`

  and `Documents`

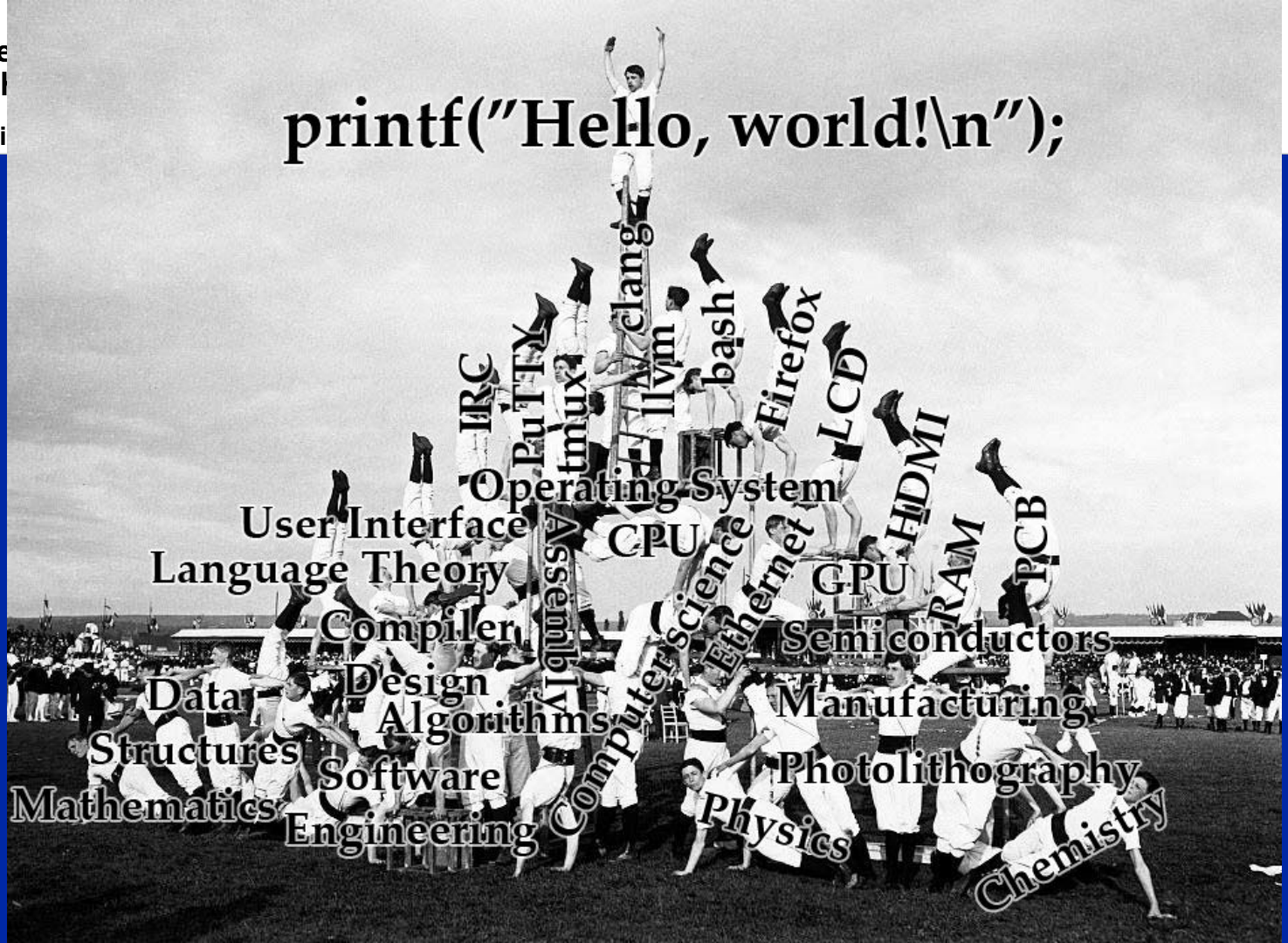
  `cd Doc` + tab will autocomplete to `cd Documents`

  `mkdir` MaKe DIRectory
- Back in the day memory space was limited and commands could only be 7 characters long

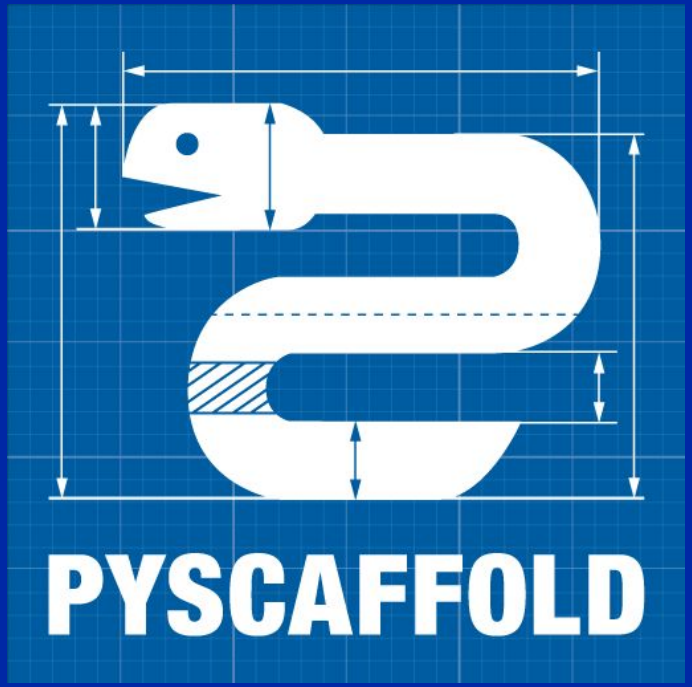
The PATH

- It’s where you computer checks for programs
- Example: /Users/kbuban/PycharmProjects/appb/venv/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/TeX/texbin:/opt/X11/bin
- The terminal checks all of these folders for a file named like the command you just typed and executes it
- `which <program_name>` shows you where a program is stored on you computer
- E.g. `which git` shows you `/usr/bin/git` on unix
- If you get the message can’t find the command when trying to execute a program, check that the folder where the program was installed is in the PATH
- `echo $PATH` on Unix
- On Windows there are many ways, one is: `$Env:Path`

The tip of the iceberg


printf("Hello, world!\n");
IRC
PuTTY
tmux
Assembly
clang
llvm
bash
Firefox
LCD
HDMI
Operating System
User Interface
CPU
Language Theory
Compiler
Data
Design
Algorithms
Structures
Software
Mathematics
Engineering
Computer science
Ethernet
GPU
RAM
PCB
Semiconductors
Manufacturing
Photolithography
Physics
Chemistry

Best practices for structuring your Python project



Git

- Git is a version control system(VCS). It keeps track of versions of files. It is used with text files and not for binary data like images, video, music etc.
- Git is a deep topic and we will cover only the basics of how to install it and use some info in git.
- Download and install git: https://git-scm.com/downloads. Then use below commands.
- FIRST_NAME

- Set your username:
  git config --global user.name "FIRST_NAME LAST_NAME"

- Set your email address:
  git config --global user.email "MY_NAME@example.com"

What is a project?

- A project is a folder with a certain structure
- Can be created in different ways
  - Pycharm -> New project: This way just creates a folder with 1 file called main.py in it
  - `putup project_name`: This way creates all the structure we have seen before
  - Pycharm -> Open -> Select existing folder: This has whatever structure was beforehand in that folder already existing

Create a project scaffold

- `pip install PyScaffold`: more infos under https://pypi.org/project/PyScaffold/
- `cd ..` go to where you want install your project, any folder e.g. Documents
- `cd ..` goes a folder up
- `cd` goes to the home folder
- `cd folder_name` enters the folder called folder_name
- `pwd` shows the current working directory
- `Putup appi` creates project scaffold(directory and files)
- The putup command might fail if git was not installed or initialized, in that case install git and initialize the email and name. Putup will show you the exact commands
- `cd appi`
- `pip install -e .` inside the project directory(initializes the project, makes it read to be installed as a module)(recommended way to install package)
- `python setup.py install` to install the package(older way to install a package)
- Note: Windows user might have to activate their env in the console using: `. \venv\Scripts\activate.ps1` Or if security issues arise `.\\venv\Scripts\activate.ps1`

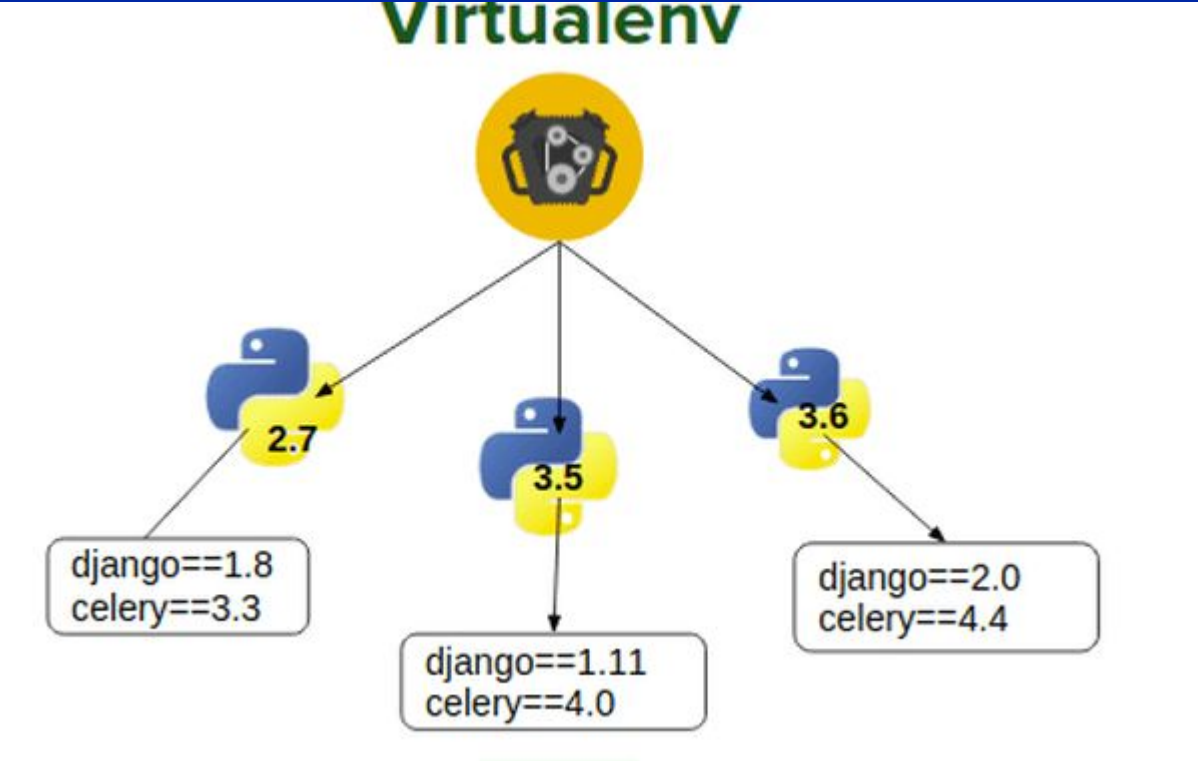
Project structure

- Build: folder where your library is being installed to
- Dist: folder where your .egg/.whl is being installed to(‘pip wheel .’ to create a single installable file)
- Docs: Documentation
- Src: Source code
- Tests: Tests
- Venv: Virtual environment
- .gitignore: what files to ignore for git
- Authors.rst: Authors
- Readme.rst: A description of your project
- Setup.py: Run this file to install your package

Project structure(advanced)

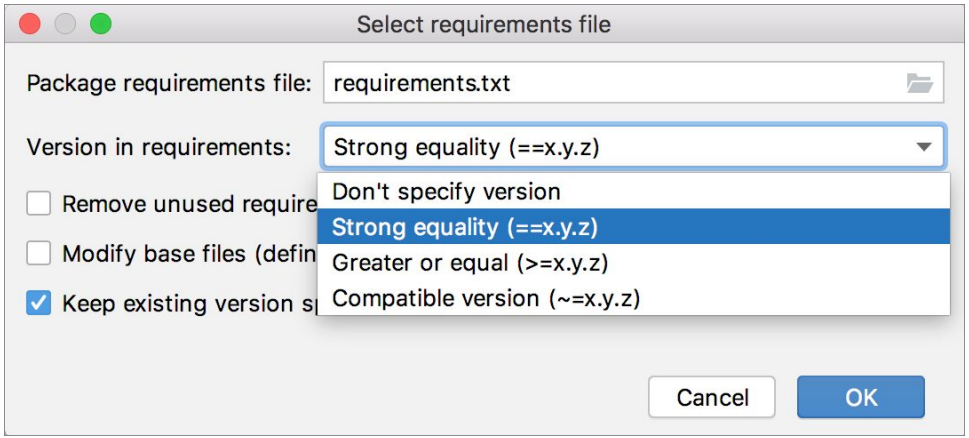
- Setup.cfg: Configuration file of how your package should be installed
- .coveragerc: Configuration file of what code to test and count towards coverage
- .readthedocs.yml: Host you documentation online
- Changelog.rst: Changes between versions
- Contributing.rst: How to contribute to the project in case it is open source
- License.txt: The software licence
- Pyproject.toml: Configuration file of what tools are needed to build your package
- Tox.ini: Configuration file of how to configure the tests for different environments

Environments and Reproducibility



Venv and requirements.txt

- Virtual environments are a way to keep different versions of python and python packages separated from each other
- Each project can use their own set of software versions
- No conflicts between projects which need different versions
- Requirements.txt allows us to specify which versions we want
  - Requests == 1.1.1(exact)
  - Matplotlib >= 1.2.1(newer)
  - Pip ~= 1.3.1(compatible)
  - Tensorflow(newest)



Semantic versioning 2.0.0

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backwards compatible manner, and
- PATCH version when you make backwards compatible bug fixes.
- https://semver.org/

Venv creation and and activation

- Create a virtualenv: https://www.jetbrains.com/help/pycharm/creating-virtual-environment.html
- In Pycharm when you open the terminal if a venv is associated with the current project it is activated automatically on Unix systems, on Windows you need to activate it manually.
- To manually activate a venv you need to source the activation script
- `source script` runs a script which configures your environment
- `. Script` dot is a shortcut for the `source` command
- When you activate a virtual environment you source the activate script
- `. venv/bin/activate` on Unix
- On Windows: `. \venv\Scripts\activate.ps1` or if security issues arise `.\\venv\Scripts\activate.ps1`
- You know that you are in an activated virtual environment when you see (some_name) in front of your terminal, most of the time this will be called (venv), you could name it differently, but don't
- `Deactivate` to deactivate an environment

Universität Zürich[UZH]

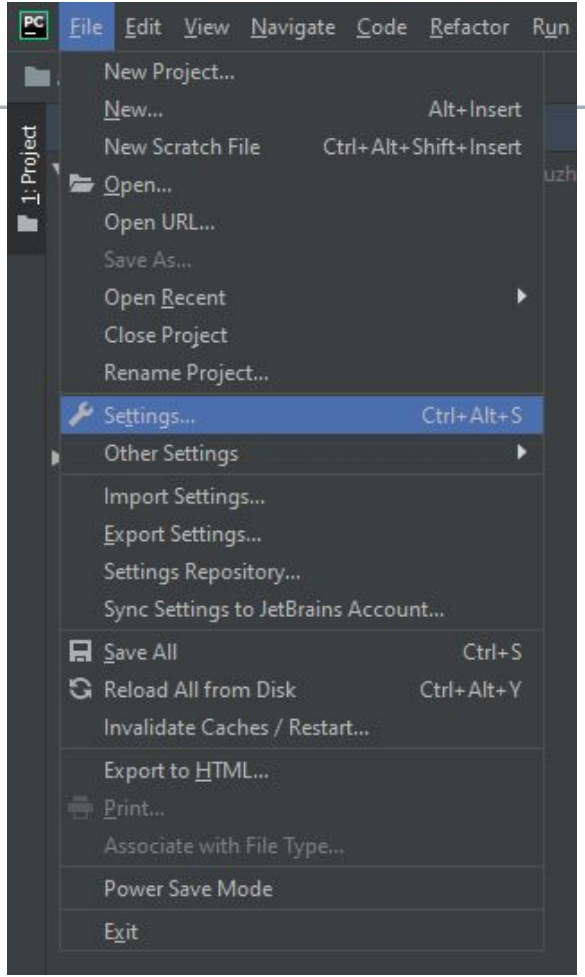
IT Training and Continuing Education

Creating a virtual environment in Pycharm

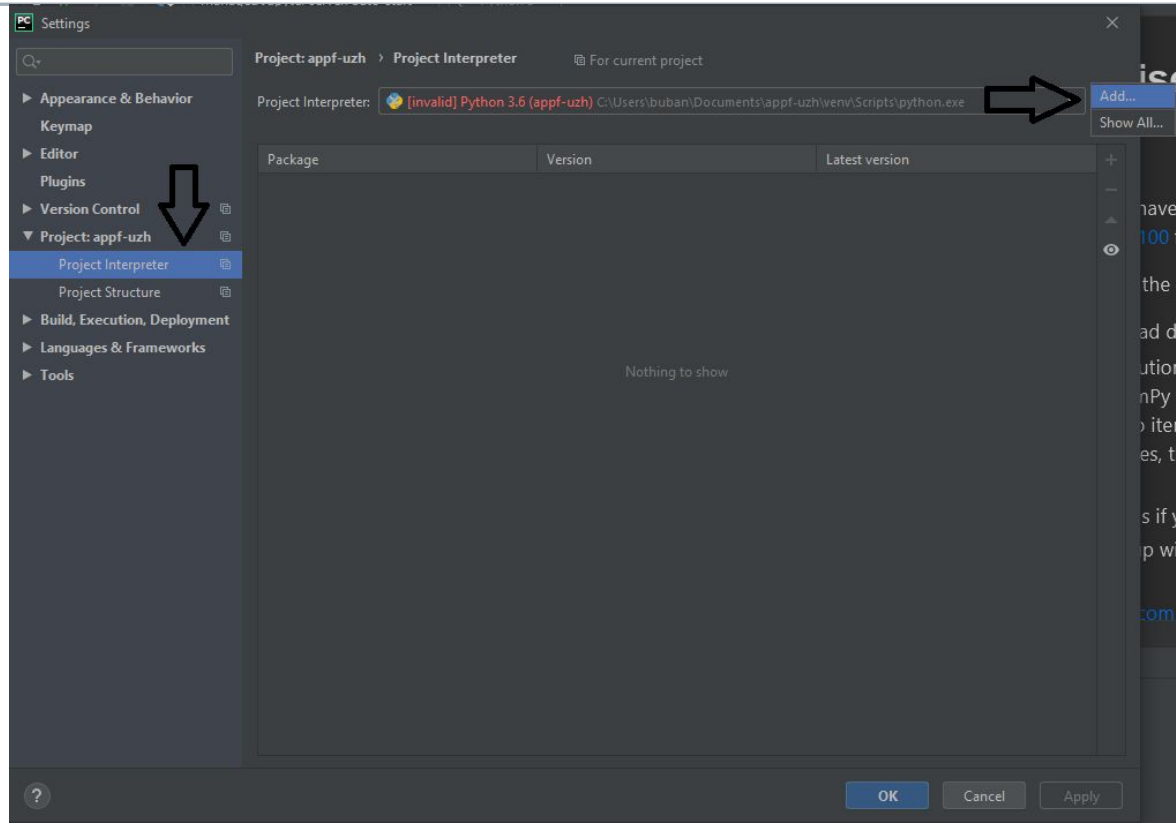
Settings

Windows: File->Settings

Mac: Pycharm->Preferences

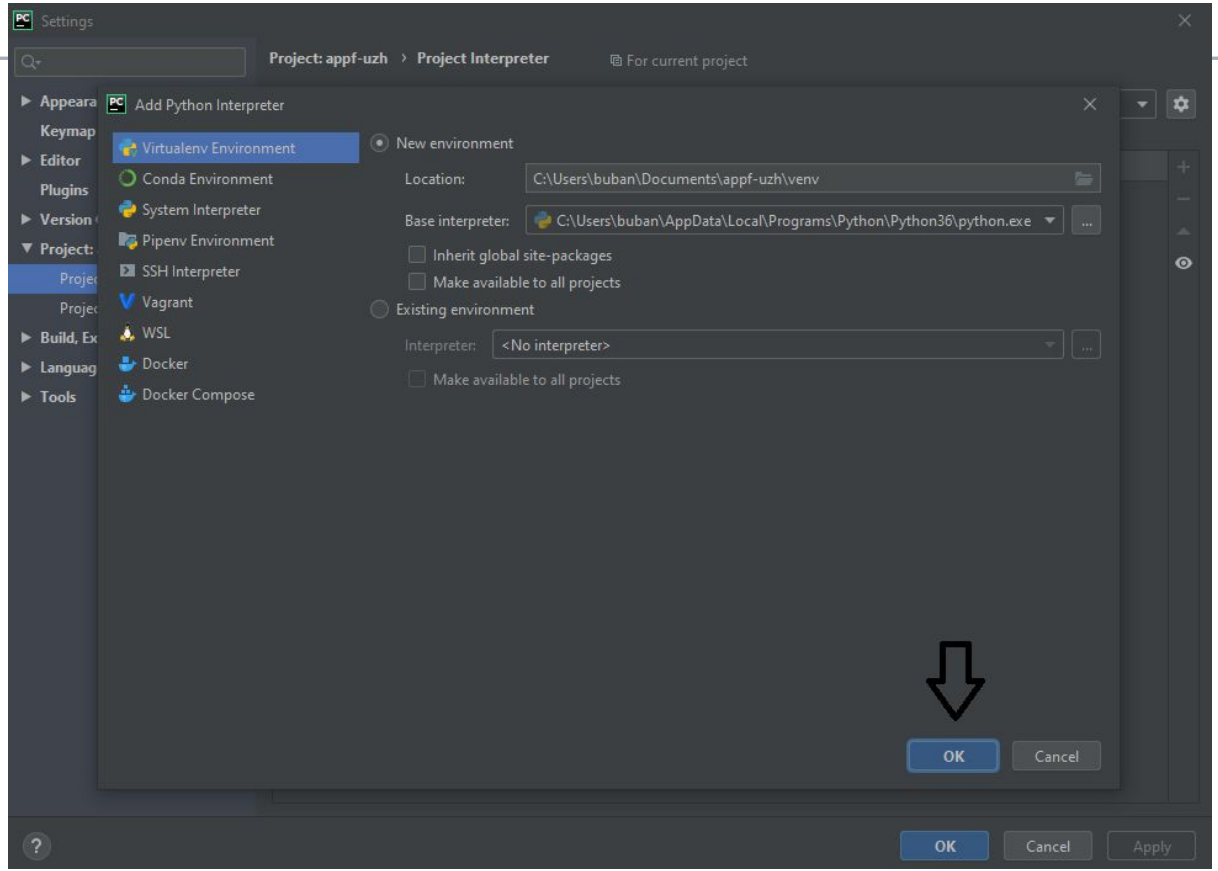


Project interpreter

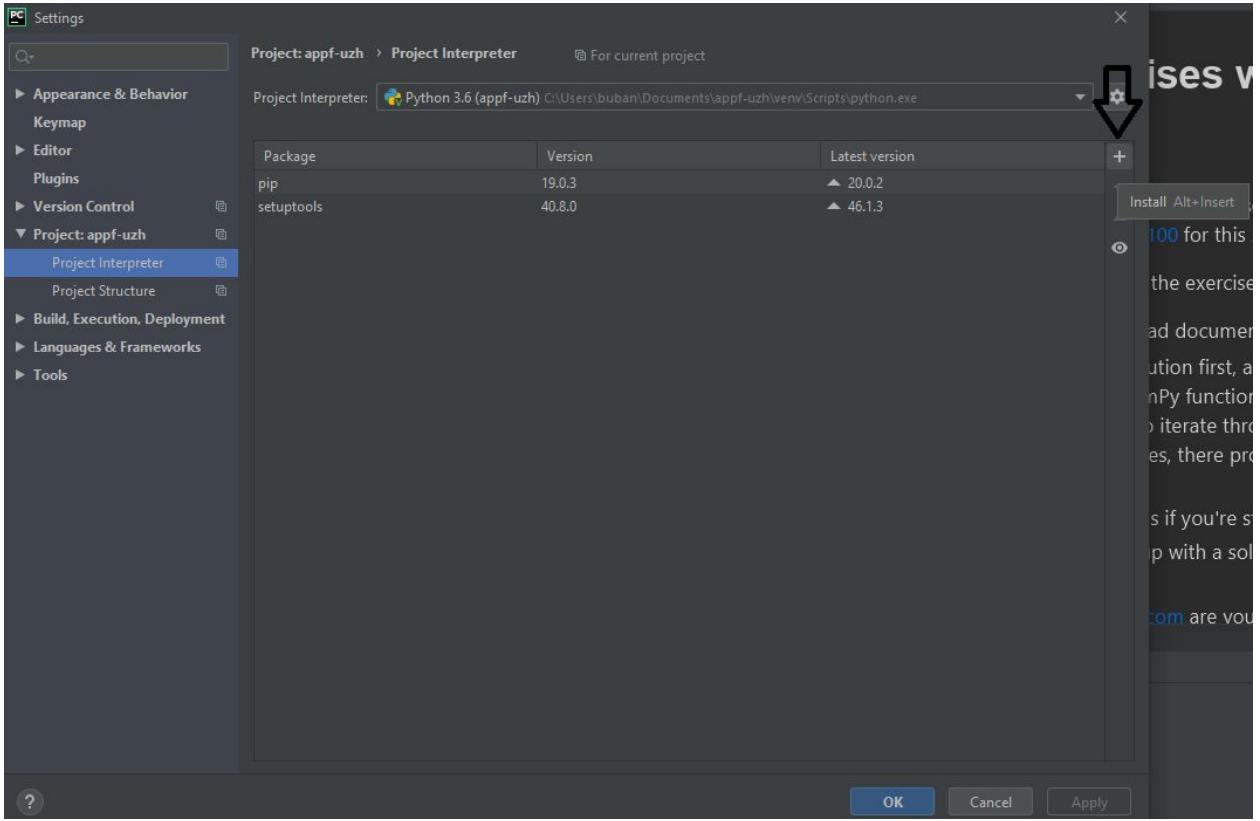


Create environment

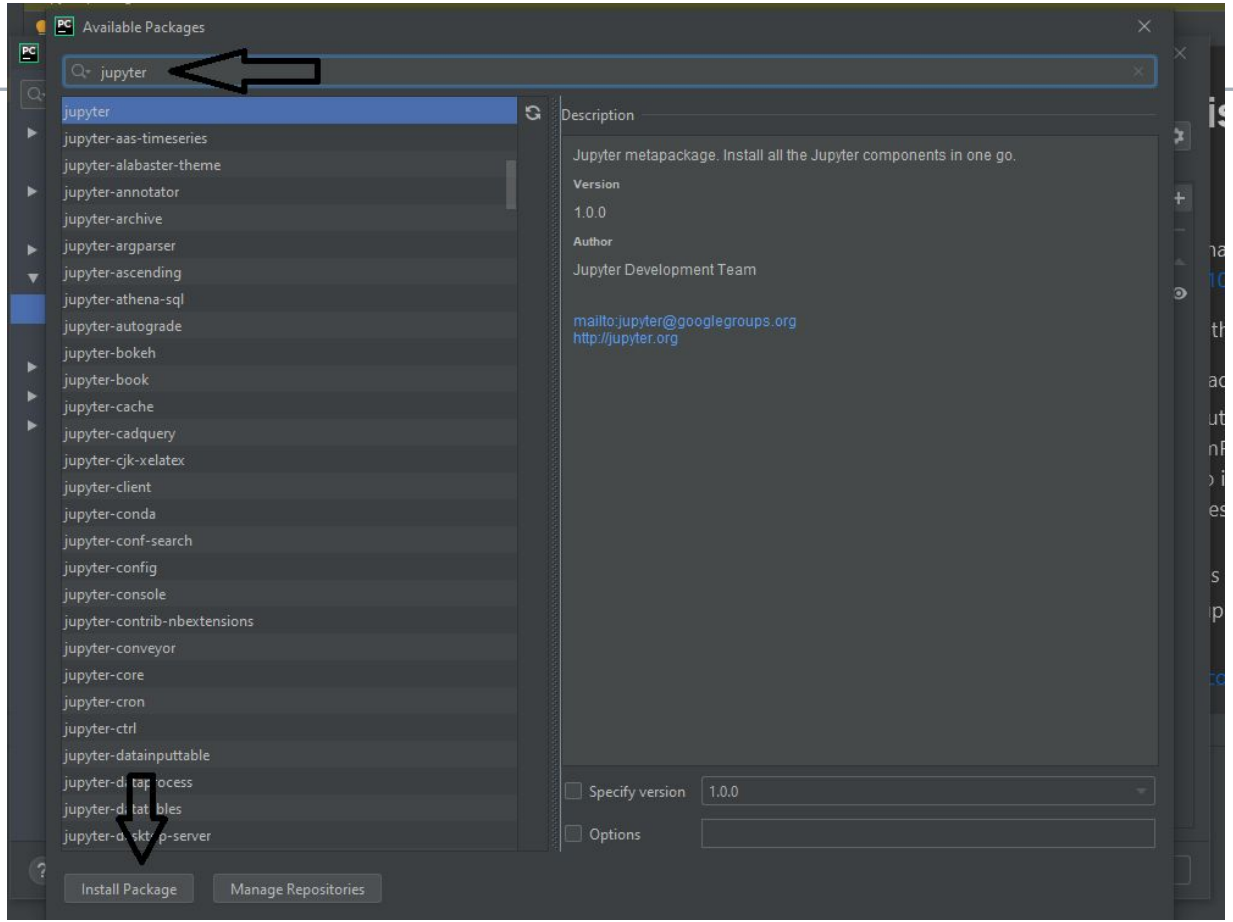
The base interpreter version doesn't matter much, if it is newer or equal to 3.6



Add packages

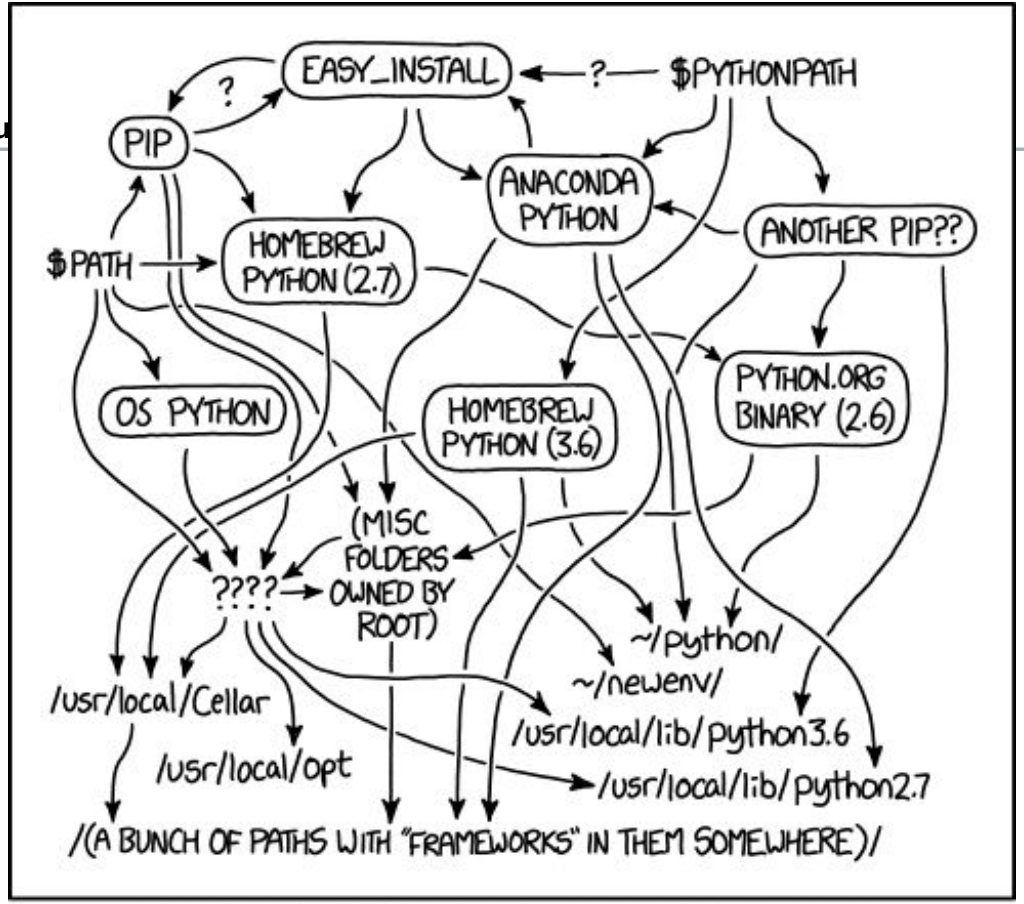


Install packages



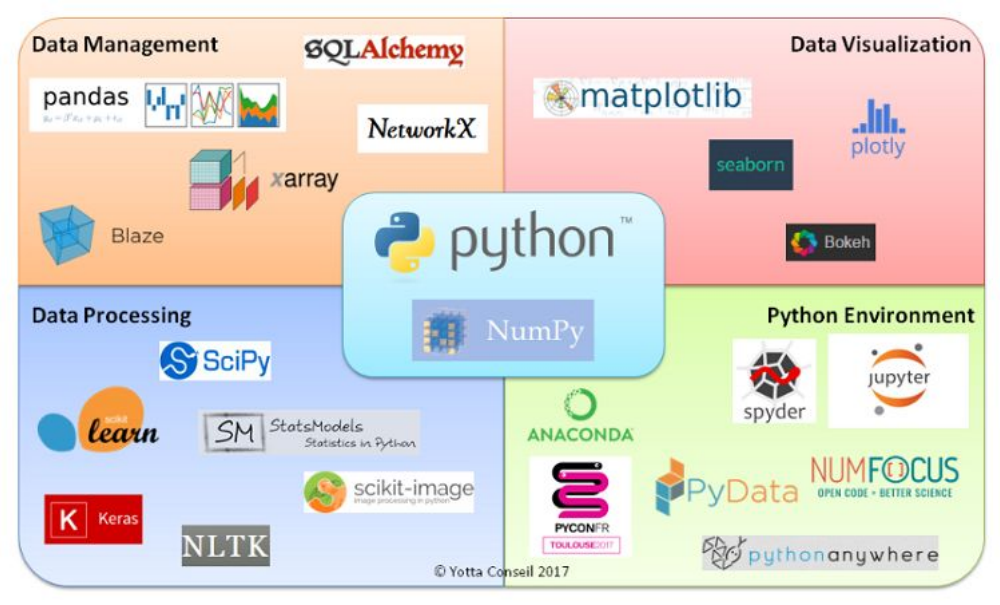
Coding time

- Create a sample project with PyScaffold
- Create a virtualenv



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED
THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

Installing and importing of packages



Finding and installing packages

- Pipy - The Python package index: https://pypi.org/
- Unofficial Windows Binaries for Python Extension Packages: https://www.lfd.uci.edu/~gohlke/pythonlibs/
- Pip install package_name
- Python setup.py develop/install “develop” is the old way, new way is -e
- Check your packages under Python Packages or Settings Python Interpreter

Coding time

– Install the package “requests”

Importing modules

- If you want to use Code from another file you need to import it
- The `import` statement copy pastes the whole file content where the `import` statement is in your file
- For example: `import math`
- By convention imports are always all the way at the top of a file
- To use functions from a module you can write `module_name.function_name(arguments)`
- For example: `math.sin(10)`

The __name__ variable

- There are special predefined variables which start with __
- When running a python file, the interpreter sets the __name__ variable to the string “__main__”
- But when the same file is being imported the __name__ variable is set to the name of the file without the file ending
- a.py
- print(__name__) # prints “__main__”
- b.py
- import a # prints “a”

The main() function

- By convention the code which is specific to your current project resides in the main function
- Generic code which can be reused in other projects doesn't reside in the main function but in their own functions
- This convention has two benefits:
  - When a file is imported nothing is executed without the user requesting so
  - The global namespace of the file stays empty and naming conflicts are less likely

Check out below example and skeleton.py

a.py

```
print(5)
print("Hallo world")



# Output
# 5
# Hallo World
```

b.py

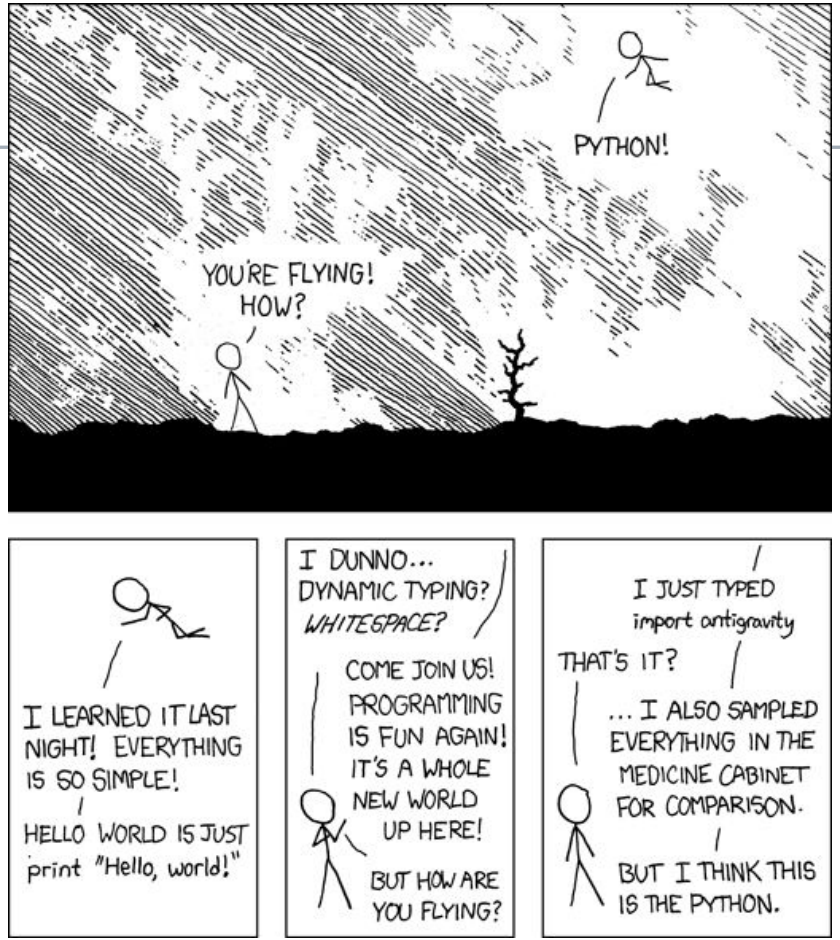
```
def my_f(a):
    print(a)


def my_f2(name):
    print("Hallo " + name)


def main():
    my_f(5)
    my_f2("World")


if __name__ == "__main__":
    main()

# Output
# 5
# Hallo World
```

c.py

```
import a
# Output
# 5
# Hallo World


import b
# Output
# Nichts

b.my_f(8) # 8
b.my_f2("Lisa") # Hallo Lisa
```

Python

– Xkcds <3



*args and **kwargs(arguments and keyword arguments)

sample_kwargs.py > ...

```python
import json

def greet(**users):
    for key,value in users.items():
        print(f'{key} => {value}.')

def main():
    greet(user='Tom', city='London', pet=['Dog', 'Cat', 'Fish'])

if __name__ == '__main__':
    main()
```

fstrings

Fstrings are a convenient way to generate strings.

There are multiple ways to combine variables with strings

- x = “Tom”
- n = 20
- y = “Hallo “ + x + “.I am “ + str(n) + “ years old”
- y = “Hallo {}. I am {} years old”.format(x, n)
- y = “Hallo {name}. I am {age} years old”.format(name=x, age=n)
- y = f“Hallo {x}. I am {n} years old”

*args and **kwargs

args stands for arguments

kwargs stands for key word arguments

def foo(a, *args, **kwargs):

foo(“a”, “b”, “c”, foo=”bar”, zee=”zorg”)

- You can not have unnamed arguments after named arguments
- “a” gets passed to a
- “b” and “c” get passed into args
- foo and zee get passed to kwargs

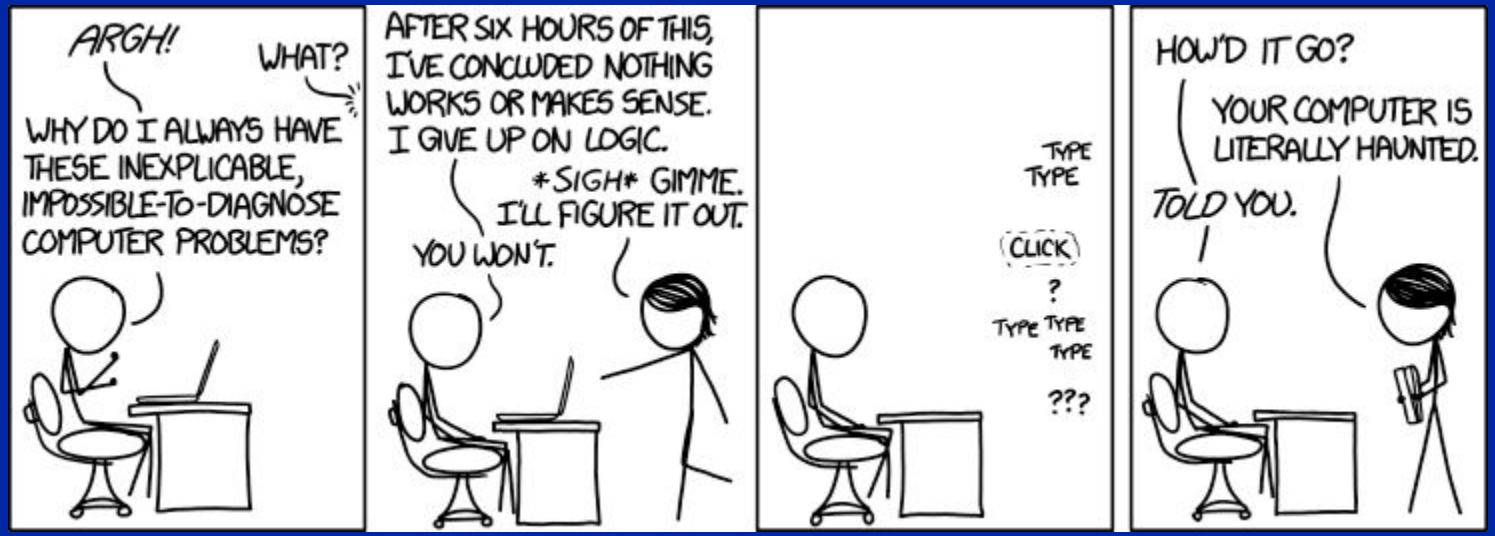
Download the project

- In Pycharm go to git->clone and enter following url: https://github.com/Kaju-Bubanja/APPI
- Watch out that the project directory is not named the same as your previously created appi folder
- Open in This Window
- Confirm the create virtual env popup

Coding time

- Write a function which takes any amount of arguments and named arguments and concatenates all the strings it finds and sums all the numbers it finds and returns both results
- Skip datatypes which are neither of those above
- Hint: Use isinstance(object, str) to find the type of an object
- Example input:
- print(foo("Hallo ", "World", 42, 10, ["Wow", 3.141], first=" Zürich", second=18, special_arg=["Special Wow", 3], last=0.333))
- Expected output:
- ('Hallo World Zürich', 70.333)

Exceptions and Debugging

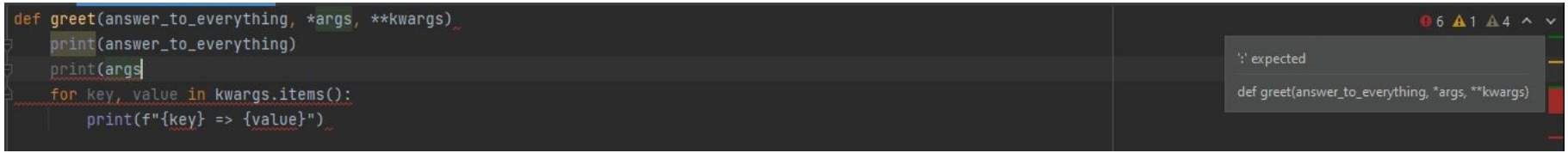


Errors: Syntax errors and exceptions

- There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.
- Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:
- >>>
- >>> while True print('Hello world')
- File "<stdin>", line 1
- while True print('Hello world')
- ^
- SyntaxError: invalid syntax
-
- The parser repeats the offending line and displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the function print(), since a colon (':') is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

IDE

- For finding errors in general, but syntax errors in specific an IDE is indispensable
- Syntax errors will show on the right and as the red squiggly lines



Exceptions

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal. Most exceptions are not handled by programs, but programmers.

Semantic errors

- Even if a statement causes no syntax error or exception it might still be incorrect
- These are the hardest errors to find
- def average(a, b):
    result = a + b / 2
    return result

Debugging

- There are many debugging techniques the most common ones are
- Read the whole code/sections of the code again line by line
- Add more logging(we will come to logging later)
- Reproduce a problem in the debugger
- Google your error ;)

Exceptions

– The most important thing when debugging is to understand the error message also called the stacktrace

```
def average(a, b):
    result = (a + b) / 2
    print(result)

a = input("Enter a number")
b = input("Enter a number")
average(a, b)
```

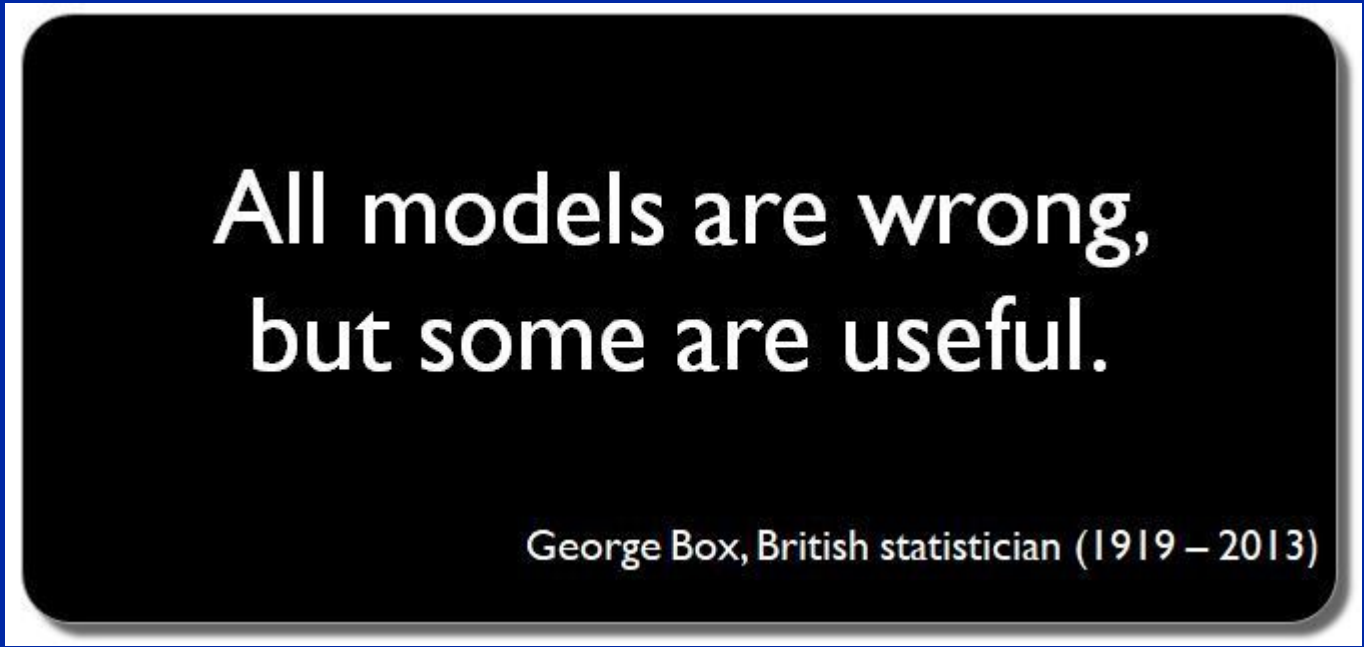
```
average()
```

```
Run:  scratch

Enter a number5
Enter a number9
Traceback (most recent call last):
  File "C:/Users/           /.PyCharm2018.2/config/scratches/scratch.py", line 7, in <module>
    average(a, b)
  File "C:/Users/           /.PyCharm2018.2/config/scratches/scratch.py", line 2, in average
    result = (a + b) / 2
TypeError: unsupported operand type(s) for /: 'str' and 'int'

Process finished with exit code 1
```

Debugger demo

- I will show you a few features of the debugger, follow along in your own

Object oriented programming(OOP)



Programming paradigms

- imperative in which the programmer instructs the machine how to change its state,
  - procedural which groups instructions into procedures,
  - object-oriented which groups instructions with the part of the state they operate on,
- declarative in which the programmer merely declares properties of the desired result, but not how to compute it
  - functional in which the desired result is declared as the value of a series of function applications,
  - logic in which the desired result is declared as the answer to a question about a system of facts and rules,
  - mathematical in which the desired result is declared as the solution of an optimization problem
  - reactive in which the desired result is declared with data streams and the propagation of change

Classes and objects

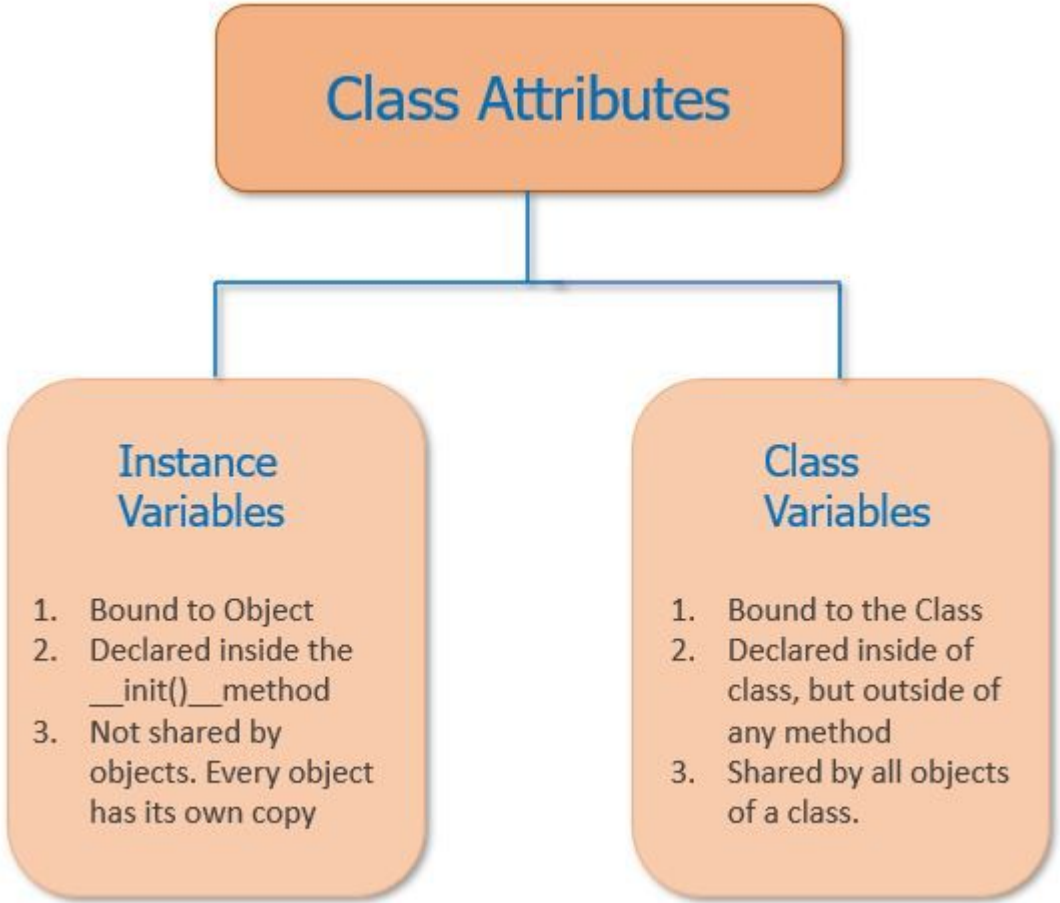
- **Class**: The class is a user-defined data structure that binds the data members and methods into a single unit. Class is a **blueprint or code template for object creation**. Using a class, you can create as many objects as you want.
- **Object**: An **object is an instance of a class**. It is a collection of attributes (variables) and methods. We use the object of a class to perform actions.

Defining a class

```
class Person:
    def __init__(self, name, sex, profession):
        # data members (instance variables)
        self.name = name
        self.sex = sex
        self.profession = profession

    # Behavior (instance methods)
    def show(self):
        print('Name:', self.name, 'Sex:', self.sex, 'Profession:', self.profession)

    # Behavior (instance methods)
    def work(self):
        print(self.name, 'working as a', self.profession)
```

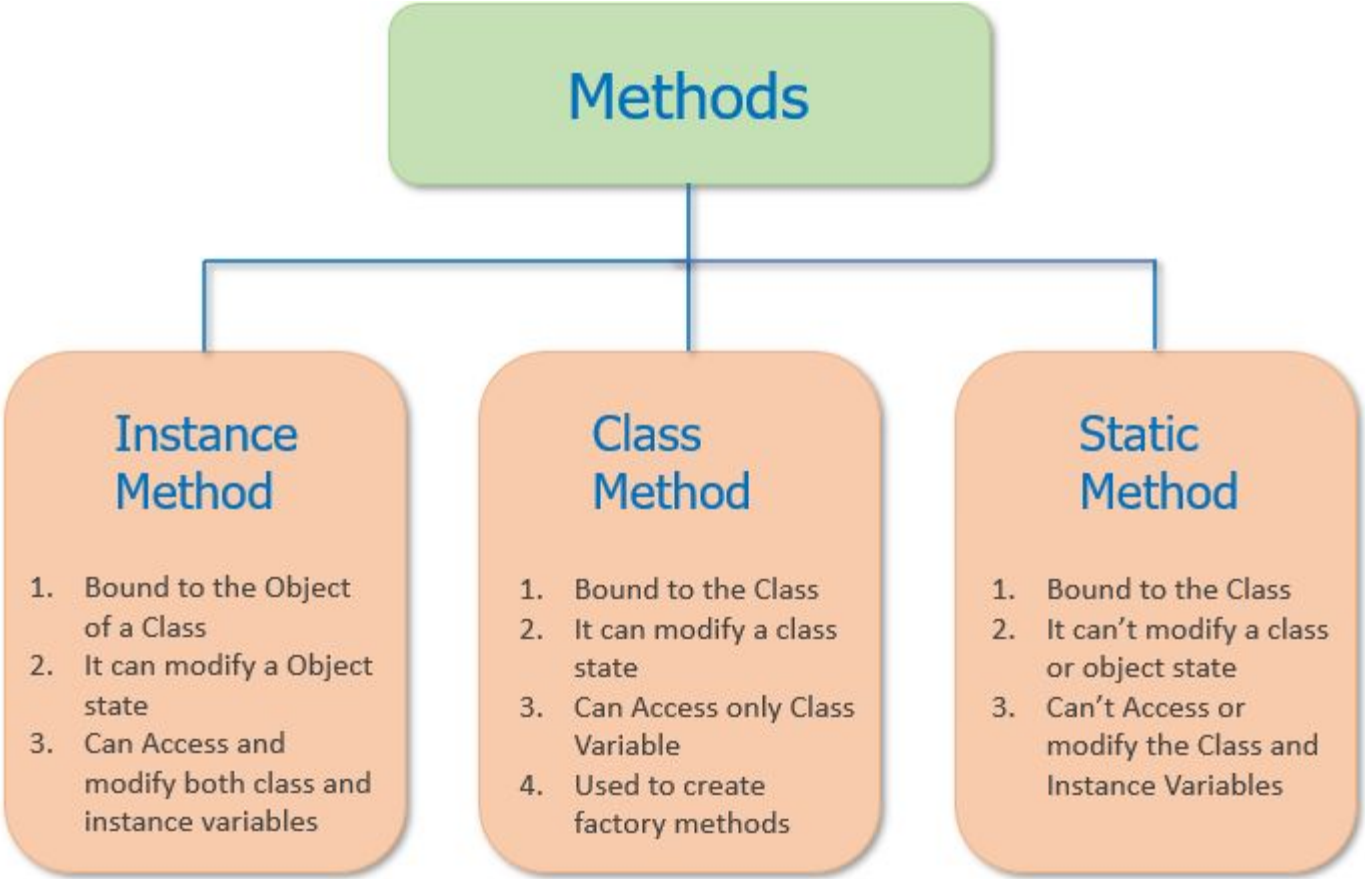
Creating objects

- <object-name> = <class-name>(<arguments>)
- jessa = Person('Jessa', 'Female', 'Software Engineer')
- jessa.show()
- jessa.work()


Class Attributes
Instance Variables
1. Bound to Object
2. Declared inside the __init()__method
3. Not shared by objects. Every object has its own copy
Class Variables
1. Bound to the Class
2. Declared inside of class, but outside of any method
3. Shared by all objects of a class.

Examples

- Check out person.py and class_attributes.py


Methods
Instance Method
1. Bound to the Object of a Class
2. It can modify a Object state
3. Can Access and modify both class and instance variables
Class Method
1. Bound to the Class
2. It can modify a class state
3. Can Access only Class Variable
4. Used to create factory methods
Static Method
1. Bound to the Class
2. It can't modify a class or object state
3. Can't Access or modify the Class and Instance Variables

Attribute lookup order

- If the same attribute name occurs in both an instance and in a class, then attribute lookup prioritizes the instance:

```
class Warehouse:
        purpose = 'storage'
        region = 'west'

        def __init__(self):
            self.region = "east"

w1 = Warehouse()
print(w1.region)
# prints "east"
print(Warehouse.region)
# prints "west"
```

On the fly attribute creation

```
class Warehouse:
        purpose = 'storage'
        region = 'west'
```

```
w1 = Warehouse()
print(w1.region)
# output: west
w2 = Warehouse()
w2.region = 'east'
print(w2.region)
# output: east
print(w1.region)
# output: west
print(Warehouse.region)
# output: west
```

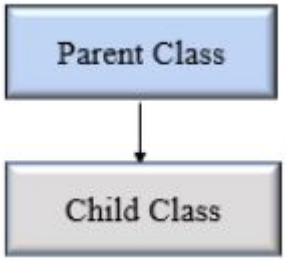
Class naming convention

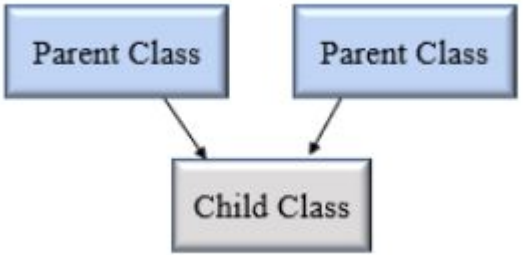
Naming conventions are essential in any programming language for better readability. If we give a sensible name, it will save us time and energy later. Writing readable code is one of the guiding principles of the Python language.

- Class names should follow the **UpperCaseCamelCase** convention
- Exception classes should end in "**Error**".
- Python's built-in classes are typically lowercase words

Inheritance

- Single inheritance
- Multiple inheritance
- Mixins





- Classes should inherit from a parent class if they have a “is a” relationship e.g. Dog is an Animal.
- If unsure if x is a y, favor composition over inheritance e.g. Kitchen “has an” oven.
- For is a relationships use inheritance
- For has a relationships use composition aka class or instance attributes

Super and method overriding

- When a class inherits all properties and behavior from the parent class is called inheritance. In such a case, the inherited class is a subclass and the latter class is the parent class.
- In child class, we can refer to parent class by using the super() function. The super function returns a **temporary** object of the parent class that allows us to call a parent class method inside a child class method.

Benefits of using the super() function.

1. We are not required to remember or specify the parent class name to access its methods.
2. We can use the super() function in both **single** and **multiple inheritances**.
3. The super() function support code **reusability** as there is no need to write the entire function
   - Check out the example parent_methods.py

Printing classes

- By implementing the `def __str__(self):` method you can decide how the class shows up when you use print(object). The method needs to return a str
- There is a subtle difference between __repr__ and __str__. For more details check: https://stackoverflow.com/questions/1436703/what-is-the-difference-between-str-and-repr/2626364#2626364
- To check what type an object is you can use print(type(object))

Coding time

- Solve the exercises in the oop/exercises.py file
- All the exercises are independent from each other

Classes real usage example - Interfaces

- We have a device which uses a modem to send messages
- We implement 3 different modems from 3 suppliers
- We have one BaseClass which acts as an interface, a contract which all subclasses need to fulfill
- class Modem:

```
def __init__(self, name):
    self.name = name
def create_msg(self, error):
    log.warning(f“Not implemented for this modem type {self.name}”)
def send_msg(self, msg, recipient):
    log.warning(f“Not implemented for this modem type {self.name}”)
```

Classes real usage example - Subclass

```
Class ModemA(Modem):
    def __init__(self, name):
        super().__init__(self, name)
    def create_msg(self, error):
        # This is different for each subclass
        return f“Following error happened {error}
    def send_msg(self, msg, recipient):
        # This is different for each subclass
        call_function_to_send_msg(msg, recipient)
```

```
Class ModemB(Modem):
    …..
```

```
Class ModemC(Modem):
    …..
```

Classes real usage example - Usage

```
from somewhere import ModemA, ModemB, ModemC

def main():
        which_modem = “b”
        recipient = “0782134148”
        if which_modem == “b”:
                modem = ModemB(“b”):
        while True:
                some_condition():
                        Break
                error = check_if_error_happened()
                msg = modem.create_msg(error)
                modem.send_msg(msg, recipient)
```

Handling exceptions and logging

Handling exceptions

- It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using Control-C or whatever the operating system supports); note that a user-generated interruption is signalled by raising the KeyboardInterrupt exception.
- >>>
- >>> while True:
- ... try:
- ... x = int(input("Please enter a number: "))
- ... print(x)
- ... break
- ... except ValueError:
- ... print("Oops! That was no valid number. Try again...")

Try ... except

The try statement works as follows.

- First, the *try clause* (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then, if its type matches the exception named after the except keyword, the *except clause* is executed, and then execution continues after the try/except block.
- If an exception occurs which does not match the exception named in the *except clause*, it is passed on to outer try statements; if no handler is found, it is an *unhandled exception* and execution stops with a message.

Exceptions: Else and finally

- else is run if no exception was triggered
- finally is always run. Even if you return inside the function

```
file = open('test.txt', 'w')

try:
    print("Writing to file.")
    file.write("Testing.")
except IOError:
    print("Could not write to file.")
else:
    print("Write successful.")
finally:
    file.close()
    print("File closed.")
```

Catching multiple exceptions

- A try statement may have more than one *except clause*, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding *try clause*, not in other handlers of the same try statement. An *except clause* may name multiple exceptions as a parenthesized tuple, for example:
- … except (RuntimeError, TypeError, NameError):
- … pass
- One can also catch multiple exceptions and perform a different action for each
- … except RuntimeError:
  … print(“RuntimeError”)
  … except TypeError:
  … print(“Wrong type”)
  … except NameError:
  … print(“Variable not defined”)

Logging

- Logging is a means of tracking events that happen when some software runs. The software's developer adds logging calls to their code to indicate that certain events have occurred. An event is described by a descriptive message which can optionally contain variable data (i.e. data that is potentially different for each occurrence of the event). Events also have an importance which the developer ascribes to the event; the importance can also be called the *level* or *severity*.
- Check out when to use logging and the different severity levels: https://docs.python.org/3/howto/logging.html#when-to-use-logging
- Logging goes hand in hand with exception handling
- When an exception happens often we can not handle the error, the safest and usually the solution causing the least problems is to log as much context(variables, stacktrace, resources) as possible and to abort the program

Coding time

- Check out the log_handling.py file of how to set up a logger
- Solve the exercises in the debugging_exercise.py file

File I/O(Input/Output)

Files

- Procedure to interact with files
  - Open file
  - Read/Write to it
  - Close file
- file = open(“my_file.txt”, “w”)
  file.write(“Hallo world”)
  file.close()

File modes

- *file = open('my_file.txt', mode)*
- The mode defines what happens to the file contents and where new content is written to
- Mode
  - 'r': read only
  - 'w': new file will be created for writing (existing file content will be deleted)
  - 'r+': reading and writing
  - 'a': append data at the end of the file(useful for a log file)

Context manager

- Context managers saveguard resources in case of an exception. The most widely used example of context managers is the with statement. Suppose you have two related operations which you'd like to execute as a pair, with a block of code in between. Context managers allow you to execute a pair of related operations like opening and closing a resource. For example:
- with open('some_file', 'w') as opened_file:
- opened_file.write('Hola!')
- The above code opens the file, writes some data to it and then closes it. If an error occurs while writing the data to the file, it tries to close it. The above code is equivalent to:
- file = open('some_file', 'w')
- try:
- file.write('Hola!')
- finally:
- file.close()

Custom context manager

- To create an object which can be used in a with statement you need to implement the __enter__(self): and __exit__(self): methods. The enter method needs to return self.
- Check out the example in custom_context_manager.py

Saving data to a file

- There are two main ways to save data to a file: json and pickle
  - x = {“foo”: “bar”}
  - with open(“file.txt”, “w”) as file: # pickle needs binary mode so “wb”
- The advantage of json is that it is human readable, but it can only serialize simple data types (list, dicsts)

```
json.dump(x, file)
```

- The advantage of pickle is that it can save complex data types (classes, custom classes), but it is not human readable

```
pickle.dump(x, file)
```

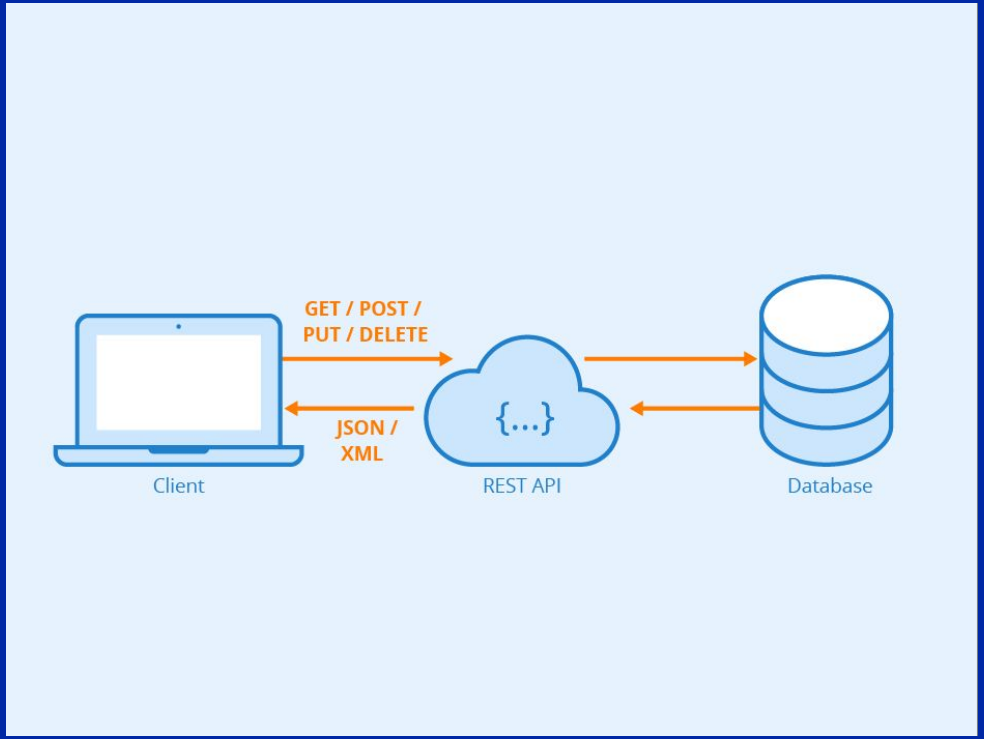
Restoring data from a file

- Both JSON and pickle can directly take a file also called file handle and restore the data
    - with open('address_book.txt', 'r') as address_book_file: # pickle needs binary mode so “rb”
- JSON
        - address_book_dict = json.load(address_book_file)
- Pickle
        - address_book_dict = pickle.load(address_book_file)

Coding time: Phonebook

- Check out the phonebook_exercise.py file

Handling web resources



Http

- The **Hypertext Transfer Protocol** (**HTTP**) is an application layer(aka websites) protocol in the Internet protocol suite model for distributed, collaborative, hypermedia information systems.
- **Hypermedia**, an extension of the term hypertext, is a nonlinear medium of information that includes graphics, audio, video, plain text and hyperlinks
- **Hypertext** is text displayed on a computer display or other electronic devices with references (hyperlinks) to other text that the reader can immediately access.
- HTTP defines methods to indicate the desired action to be performed on an identified resource.
- For our purposes, the most important thing about REST is that it's based on the four methods defined by the HTTP protocol: POST, GET, PUT, and DELETE. These correspond to the four traditional actions performed on data in a database: CREATE, READ, UPDATE, and DELETE.

URL

- **URL (Uniform Resource Locator)** - An address for a resource on the web, such as https://programminghistorian.org/about. A URL consists of a **protocol** (http://), domain (programminghistorian.org), and optional **path** (/about). A URL describes the location of a specific resource, such as a web page.
- A **query string** is a part of a uniform resource locator (URL) that assigns values to specified parameters. For example:
- https://example.com/path/to/page?name=ferret&color=purple
- It's a way to pass data from a client(browser, python script) to the server.
- **IMPORTANT**: Sensitive information should never be passed using query strings, but with a POST method and inside the body of the POST request. The main reason being that query parameters get logged in many places. For details see: https://security.stackexchange.com/questions/29598/should-sensitive-data-ever-be-passed-in-the-query-string
- TLDR of the question is **NO**!

Rest API

- An **application programming interface** (**API**) is a connection between computers or between computer programs. It is a type of software interface, offering a service to other pieces of software.
- For our purposes, the most important thing about REST is that it's based on the four methods defined by the HTTP protocol:
  - POST
  - GET
  - PUT
  - DELETE
- These correspond to the four traditional actions performed on data in a database:
  - CREATE
  - READ
  - UPDATE
  - DELETE

JSON

- **JSON (JavaScript Object Notation)** is a text-based data storage format that is designed to be easy to read for both humans and machines. JSON is generally the most common format for returning data through an API.
- Example: '{"name":"John", "age":30, "car":null}'

Databases

- A database is **an organized collection of structured information, or data**, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS) or a driver(from a script). Most databases use structured query language (SQL) for writing and querying data.
- SQL (**Structured Query Language**) is a standardized programming language that's used to manage relational databases and perform various operations on the data in them. SQL became the de facto standard programming language for relational databases after they emerged in the late 1970s and early 1980s.
- Examples:
- CREATE TABLE IF NOT EXISTS Customers (name text NOT NULL, city text, age real)
- SELECT name, city FROM Customers;
- SELECT * FROM Customers WHERE city='Mexico' AND age >= 18;
- DELETE FROM Customers WHERE name='Bob'
- INSERT INTO Customers (name, city, age) VALUES (“Bob”, “Mexico”, 25)

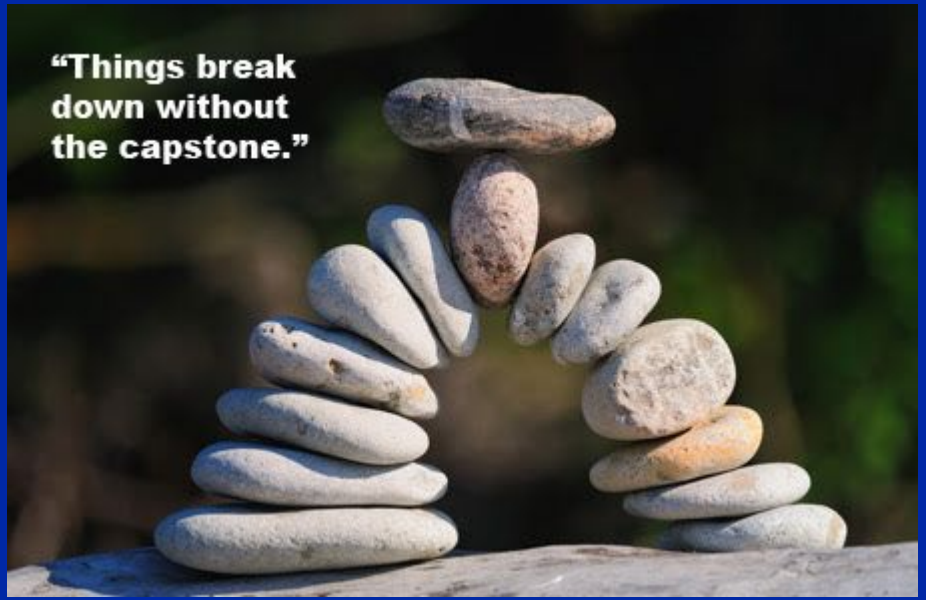
ACID

- A very important reason to use databases is because they have acid properties:
- **Atomicity**: Transactions are often composed of multiple statements. Atomicity guarantees that each transaction is treated as a single "unit", which either succeeds completely, or fails completely
- **Consistency**: Consistency ensures that a transaction can only bring the database from one valid state to another, maintaining database invariants
- **Isolation**: Transactions are often executed concurrently (e.g., multiple transactions reading and writing to a table at the same time). Isolation ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially
- **Durability**: Durability guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure (e.g., power outage or crash). This usually means that completed transactions (or their effects) are recorded in non-volatile memory

Coding time: Making our own Rest API

- Patrick Smyth provides an excellent tutorial on making our own Rest API
- https://programminghistorian.org/en/lessons/creating-apis-with-python-and-flask
- We will go through it as a preparation for our capstone project where you can apply everything you learned in the course

Capstone exercise



Inventory program

- We will write a small website where we can add/delete animals to/from an inventory list.
- We will save the data to a database
- We will build an API to be able to search the data
- We will build an API controller which allows us to interact with various APIs(optional)

- Open up the files app_exercise.py and controller_exercise.py(optional) and complete the missing functions/methods. The comments provide additional hints what the method should do. Feel free to add more functions/methods if you need

Hints for capstone exercise

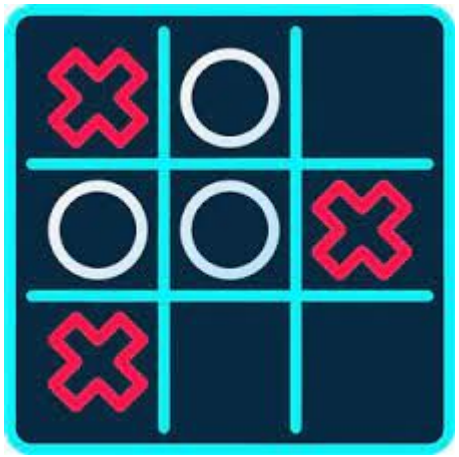
- To be able to filter numeric data not only for equality but also larger smaller etc we can use the names lt(less than), lte(less than equal), gt(greater), gte(greater equal), eq(equal) and pass these parameter in a json string. E.g. api?price={"gt": 60, “lte”: 100}&name=Bob
- APIs return some machine readable format, most commonly JSON, make your own api also return json by using jsonify on a dict. Websites return human readable content like HTML. See the index method and the main.html file for a small example

Requests(optional, used for controller)

- **Requests** is an elegant and simple HTTP library for Python, built for human beings.
- Import requests
- r = requests.get('https://api.github.com/events')
- payload = {'key1': 'value1', 'key2': 'value2'}
- r = requests.get('https://httpbin.org/get', params=payload)
- print(r.url) # https://httpbin.org/get?key2=value2&key1=value1
- But to be able to pass json(like) data we need payload = urllib.parse.urlencode(filter)
- Because json only accepts “ as quotation marks, whereas urls accept both and urlencode gives ‘. That’s why on the server side we need to replace the ‘ with “ before loading the json
- data = data.replace("\'", "\"")

Tic tac to game

- To round off our understanding of classes, objects, inheritance and interfaces
- we will program a tic tac to game
- Look at the tic_tac_to_excercise.py file and fill in the missing methods
- The computer can have a smart algorithm or just randomly chose spots, that is up to you



Wrap up

Summary

- Computer basics
- Learning to use the terminal
- Best practices for structuring your Python project
- Environments and Reproducibility
- Installing and Importing of Packages
- Functions with multiple arguments
- Exceptions and Debugging
- Basics of Object-oriented programming in Python
- A deeper dive into File I/O
- Handling web resources
- Basics of databases
- Capstone exercise (combining most of the topics)

Feedback

- After this course you will receive an email by the course direction asking for feedback about this course
- I would be more than happy to receive as much feedback as possible, since I'd love to further improve the course material and/or my teaching skills where needed
- Constructive criticism and positive comments are both very welcome
    - It's good to know where one can improve, for example by updating the course material or polishing the teaching skills in general
    - It's also good to know which parts of the course and/or which teaching skills helped you the most during the course

Further resources

- Python advanced course with focus on data analysis: Python - Data Analysis Essentials
- Stackoverflow is a great site to find information and to ask questions
- Please read the Tour and the how to ask guides first, before asking questions. This will save you from downvotes and shows respect towards people who answer questions.
- https://stackoverflow.com/tour
- https://stackoverflow.com/help/how-to-ask
- You will be surprised how many people all around the world are ready to help you out to learn to program even better :)

Thank you