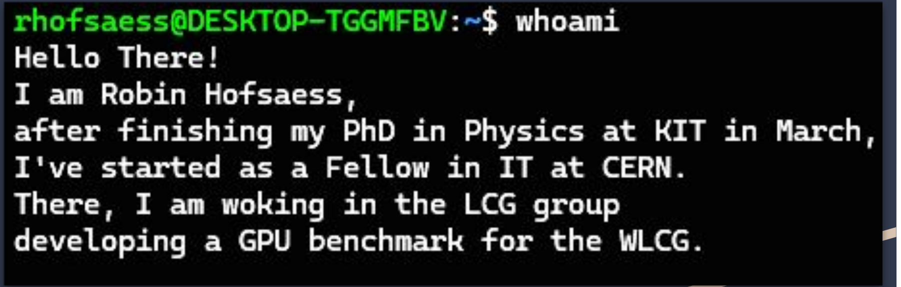
Introduction

What is Python and when to use it

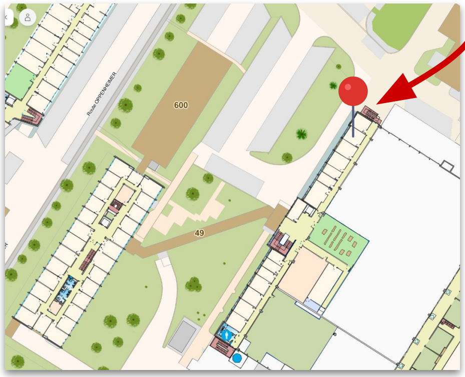
Part I



```
rhofsaess@DESKTOP-TGGMFBV:~$ whoami
Hello There!
I am Robin Hofsaess,
after finishing my PhD in Physics at KIT in March,
I've started as a Fellow in IT at CERN.
There, I am woking in the LCG group
developing a GPU benchmark for the WLCG.
```

Hi
and welcome!

I am here!



What this course is (not) about...

- **This course will not teach you how to program in Python!**
- We will focus on the underlying **concepts** of Python
- You will learn about the **Pros** and **Cons** of the language and how to get the most out of it (and when to avoid it)
- You will know how and when to use Python at best
- Tips on **how to learn** the language properly
- We will discuss **best practices** and **helpful tools**
- How to integrate **AI** in a **useful** way

Jupyter Notebook

- Parallel to the slides, a jupyter notebook is provided with examples, tips, additional information, and some small exercises/hands-on stuff
- Available on Github: https://github.com/RHofsaess/KSETA_25

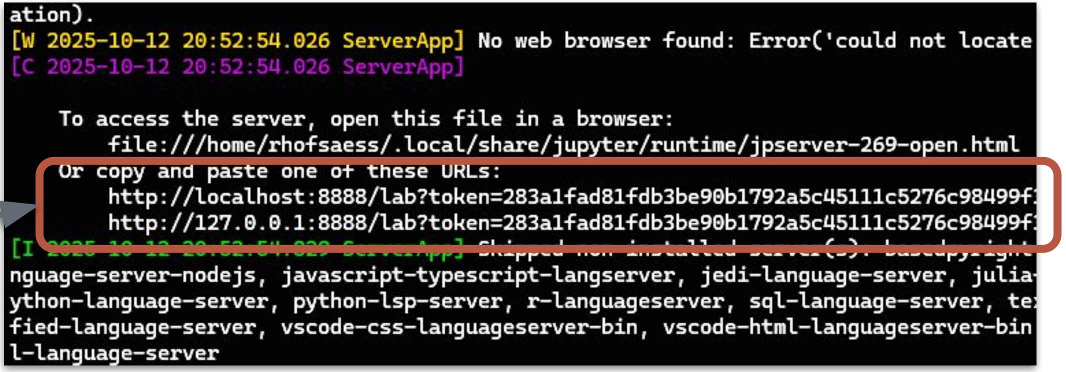
Jupyter Notebook

Let's have a look together

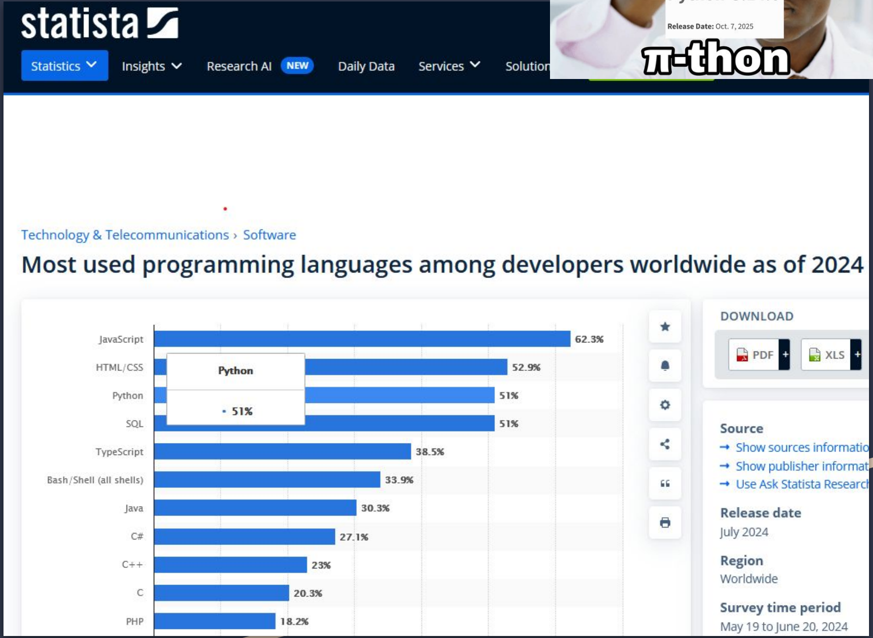
How to get started:

```
$ sudo apt install -y python3 python3-pip python3-venv

$ python3 -m venv kseta_course

$ source kseta_course/bin/activate

$ pip install jupyter

$ jupyter lab
```

Click



```
ation).
[W 2025-10-12 20:52:54.026 ServerApp] No web browser found: Error('could not locate
[C 2025-10-12 20:52:54.026 ServerApp]

    To access the server, open this file in a browser:
        file:///home/rhofsaess/.local/share/jupyter/runtime/jpserver-269-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/lab?token=283a1fad81fdb3be90b1792a5c45111c5276c98499f
        http://127.0.0.1:8888/lab?token=283a1fad81fdb3be90b1792a5c45111c5276c98499f
[I 2025-10-12 20:52:54.029 ServerApp] Skipped non-installed server(s): basedpyright
nguage-server-nodejs, javascript-typescript-langserver, jedi-language-server, julia
ython-language-server, python-lsp-server, r-languageserver, sql-language-server, te
fied-language-server, vscode-css-languageserver-bin, vscode-html-languageserver-bin
l-language-server
```



Technology & Telecommunications › Software

Most used programming languages among developers worldwide as of 2024

Source
→ Show sources informatio
→ Show publisher informat
→ Use Ask Statista Researc

Release date
July 2024

Region
Worldwide

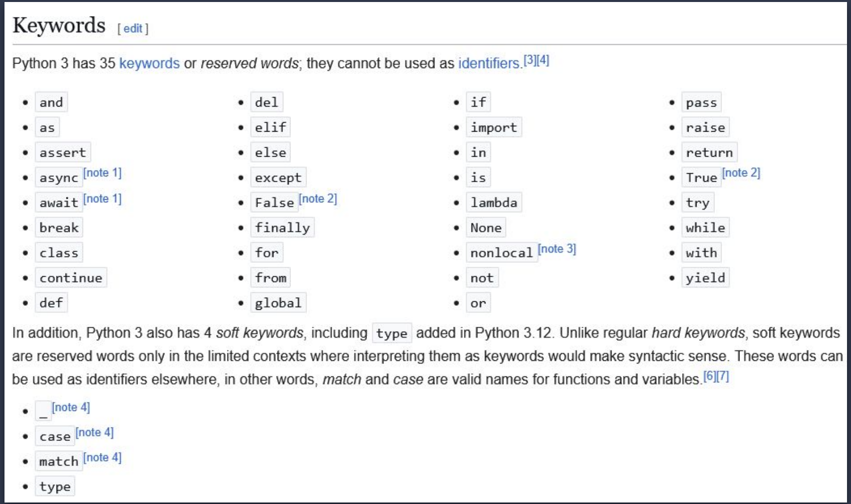
Survey time period
May 19 to June 20, 2024

Some General Information

- [Python](#) was introduced in 1991 (Python3 in 2008)
- One of the most used languages!
- Great influence, also on others
- Strongly community driven
- Constantly evolving: [PEP](#)

Technicals:

- Multi-paradigm:
  - Imperative, object-oriented, and functional
- Interpreted language
- Mainly **C(++)** under the hood
  - Reference implementation: [CPython](#)



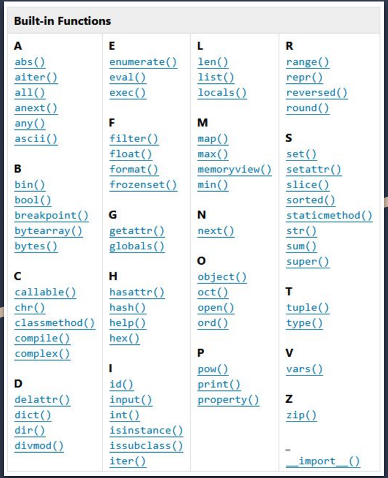
Keywords [edit]

Python 3 has 35 keywords or *reserved words*; they cannot be used as identifiers.[3][4]

- and
- as
- assert
- async [note 1]
- await [note 1]
- break
- class
- continue
- def
- del
- elif
- else
- except
- False [note 2]
- finally
- for
- from
- global
- if
- import
- in
- is
- lambda
- None
- nonlocal [note 3]
- not
- or
- pass
- raise
- return
- True [note 2]
- try
- while
- with
- yield

In addition, Python 3 also has 4 *soft keywords*, including type added in Python 3.12. Unlike regular *hard keywords*, soft keywords are reserved words only in the limited contexts where interpreting them as keywords would make syntactic sense. These words can be used as identifiers elsewhere, in other words, *match* and *case* are valid names for functions and variables.[6][7]

- _ [note 4]
- case [note 4]
- match [note 4]
- type



Built-in Functions

A
abs()
aiter()
all()
anext()
any()
ascii()

B
bin()
bool()
breakpoint()
bytearray()
bytes()

C
callable()
chr()
classmethod()
compile()
complex()

D
delattr()
dict()
dir()
divmod()

E
enumerate()
eval()
exec()

F
filter()
float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()
hash()
help()
hex()

I
id()
input()
int()
isinstance()
issubclass()
iter()

L
len()
list()
locals()

M
map()
max()
memoryview()
min()

N
next()

O
object()
oct()
open()
ord()

P
pow()
print()
property()

R
range()
repr()
reversed()
round()

S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()

The Syntax

- Python 3 consists of 35 keywords
- The syntax is close to natural language:

```
for ex in exercises:
    if not in optional:
        do exercise(ex)
    else:
        continue
```

- Additional built-in functions
- Python is a **modular** language and provides a comprehensive STL (“*Batteries Included*”)

Let’s have a short look at the syntax

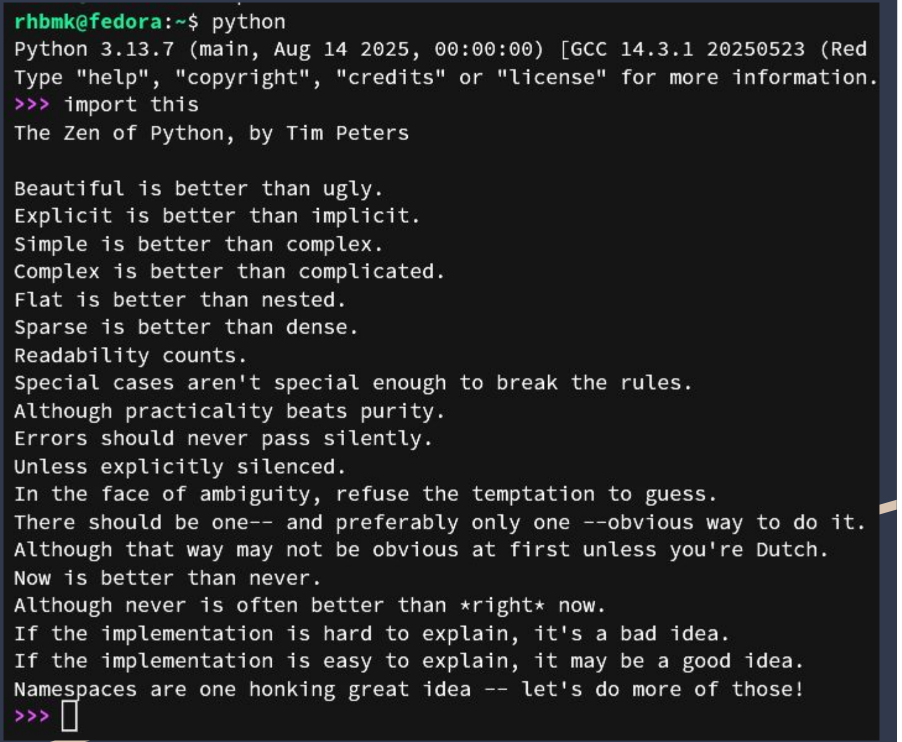
Take care! Some things can be tricky...

A very special example

The Philosophy of Python

And when to use it

Part II



```
rhbmk@fedora:~$ python
Python 3.13.7 (main, Aug 14 2025, 00:00:00) [GCC 14.3.1 20250523 (Red
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
The Zen of Python, by Tim Peters

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

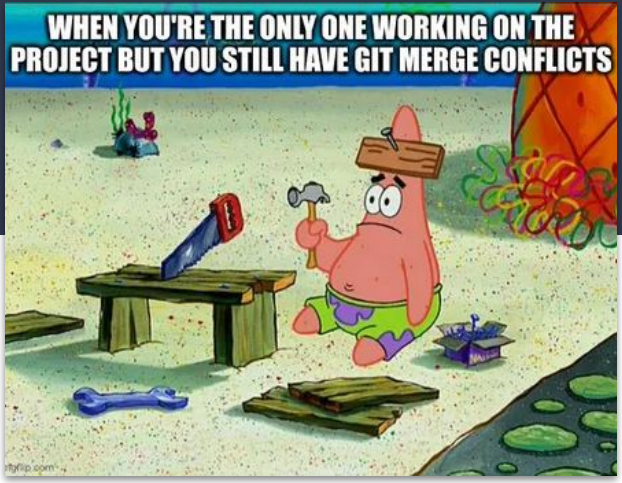
import this

The Philosophy

The *Zen of Python*:

- Good readability
- Simple is better than complex
- Consistency and clarity
- Beautiful is better than ugly
- Errors should never pass silently

Good Readability and Clarity


WHEN YOU'RE THE ONLY ONE WORKING ON THE PROJECT BUT YOU STILL HAVE GIT MERGE CONFLICTS
imgflip.com

- Good readability is key
  - Future **you** will be your own teammate!
  - Add documentation, if not directly clear
- The python style guide: PEP8
  - Use 4 spaces per indent
  - Use meaningful names: *user_input* not *inp*
  - naming conventions:
    - *snake_case* for **functions and variables**
    - *PascalCase* for **classes**
    - *ALL_CAPS* for **constants**
  - **Many more!**

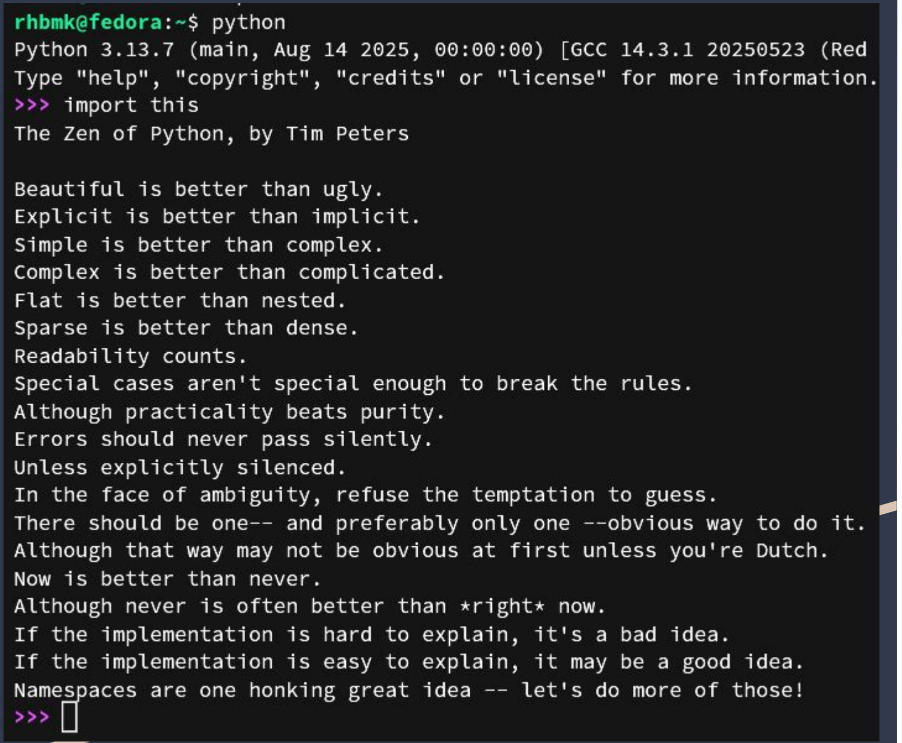
Explicit and Simple

- Helps to avoid mistakes
- Beginner tips:
    - Don’t overcomplicate things
    - Fancy is bad, easy is good
- Make the code’s intention obvious
    - At some point, you might come back to the project and have to remember!
- Performance optimization is secondary
    - But maintaining a good style is of course important!
- Use simple, minimal functions

```python
# Implicit and complex
result = process(data or default) if ready else None
```

```python
# Explicit and simple
if ready:
   if data:
        result = process(data)
   else:
        result = process(default)
else:
   result = None
```

Which implementation you prefer?


```
rhbmk@fedora:~$ python
Python 3.13.7 (main, Aug 14 2025, 00:00:00) [GCC 14.3.1 20250523 (Red
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
The Zen of Python, by Tim Peters

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

The *Zen of Python*:

- Good readability
- Simple is better than complex
- Consistency and clarity
- Beautiful is better than ugly
- Errors should never pass silently

Recommendation: Keep your code clear and simple so everyone can understand it at a glance.

Why people use Python?

- (Comparably) easy to learn, understand, and maintain
- One language for prototyp and production
- Multi-purpose: You can do nearly everything with it:
    - However, this does not mean, you should ;-)
- Huge community with tons of projects one can benefit from

How is Python used at CERN?

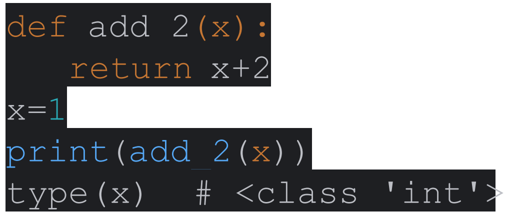
- Basically everywhere!
- As a wrapper for the experiment software (cmsRun, O2, LHCb, ...)
- For workflow management (e.g. WMAgent in CMS, luigi/law, ...)
- For task automation (we use it e.g. to feed logs into our production DB)
- For **data analysis:**
  - There are several Python-based analysis frameworks available
  - CMS:
    - CAT, ColumnFlow, Coffea-Casa, ...
- Widely used for visualizations; probably even more popular than ROOT nowadays (don't cite me on this :D)

Where Python Shines

- Prototyping - One of Python’s biggest strengths!
    - A proof of concept can be created very fast
- Learning and Development speed: Rather easy syntax
- Small/simpler ML: SciPy, PyTorch, …
- Light-weight computations and low to mid-scale **data analysis**
- Cross-platform compatibility and interoperability
- **Plotting & Visualization**
- Cross-Domain Applicability
    - like web development (Flask, Streamlit, Django), automation (Selenium), and system administration (Ansible)

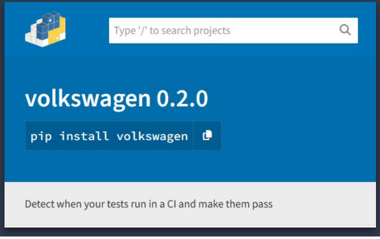
Prototyping

- **Perfect for One-shot problems**
- Dynamic typing
  - Perfect for fast prototyping
  - Python automatically resolves all types
  - The types are always accessible
- Interpreted and interactive
  - Code can directly be tested and interactively tracked
  - No need for complete implementations (as in a huge C++ framework) but a more **easy**, **step-by-step** implementation
- A working prototype is the perfect starting point for **optimizations**



```python
def add_2(x):
    return x+2
x=1
print(add_2(x))
type(x)   # <class 'int'>
```

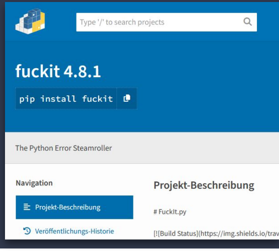
Packages, Packages Everywhere!


Type '/' to search projects

volkswagen 0.2.0

```
pip install volkswagen
```

Detect when your tests run in a CI and make them pass


Type '/' to search projects

fuckit 4.8.1

```
pip install fuckit
```

The Python Error Steamroller

Navigation

Projekt-Beschreibung

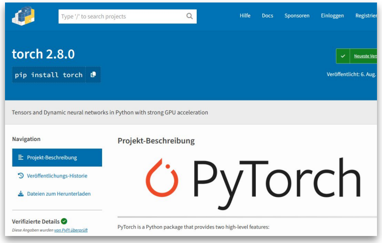
Veröffentlichungs-Historie

Projekt-Beschreibung

FuckIt.py

[![Build Status](https://img.shields.io/trav

- A huge plus for Python is the availability of countless packages
    - For most of the default problems a build-in or a community implementation exists already! **Save time and work by checking!**
- The official Python package index: Pypi
    - A small warning: Security issues with packages occurred in the past
    - Be careful and check if you find anything bad about a package
- The packages can be installed with "pip"


Type '/' to search projects

Hilfe Docs Sponsoren Einloggen Registrier

torch 2.8.0

```
pip install torch
```

Neueste Vers

Veröffentlicht: 6. Aug.

Tensors and Dynamic neural networks in Python with strong GPU acceleration

Navigation

Projekt-Beschreibung

Veröffentlichungs-Historie

Dateien zum Herunterladen

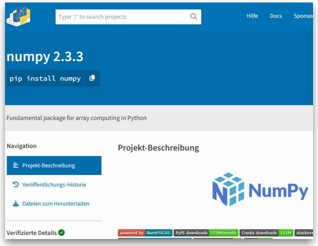
Verifizierte Details

Diese Angaben wurden von PyPI überprüft

Projekt-Beschreibung

PyTorch

PyTorch is a Python package that provides two high-level features:


Type '/' to search projects

Hilfe Docs Sponsor

numpy 2.3.3

```
pip install numpy
```

Fundamental package for array computing in Python

Navigation

Projekt-Beschreibung

Veröffentlichungs-Historie

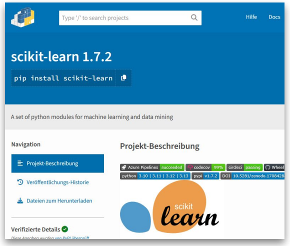
Dateien zum Herunterladen

Verifizierte Details

Projekt-Beschreibung

NumPy

powered by NumFOCUS | PyPI downloads 533M/month | Conda downloads 112M | stackove


Type '/' to search projects

Hilfe Docs

scikit-learn 1.7.2

```
pip install scikit-learn
```

A set of python modules for machine learning and data mining

Navigation

Projekt-Beschreibung

Veröffentlichungs-Historie

Dateien zum Herunterladen

Verifizierte Details

Projekt-Beschreibung

Azure Pipelines succeeded | codecov 99% | circleci passing | Wheel

python 3.10 | 3.11 | 3.12 | 3.13 | pypi v1.7.2 | DOI 10.5281/zenodo.1708428

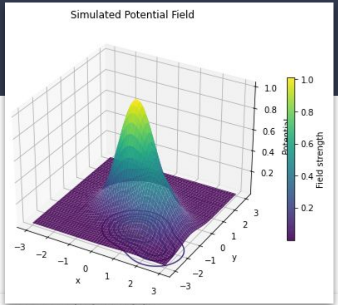
scikit learn

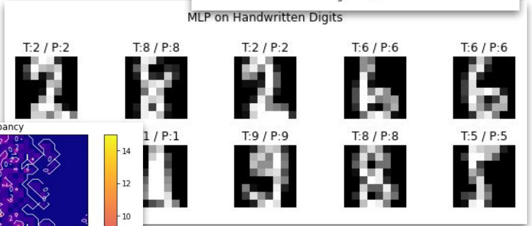
```
$ pip install <my_package>
```

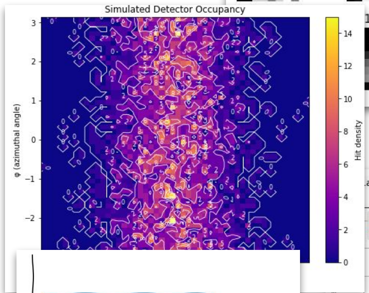
Plotting & Visualization

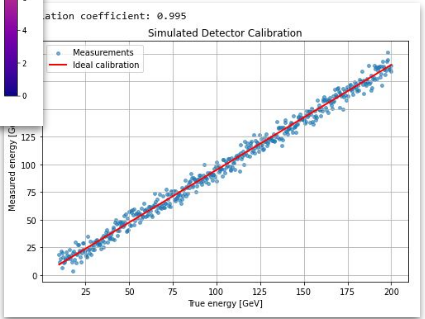
Examples

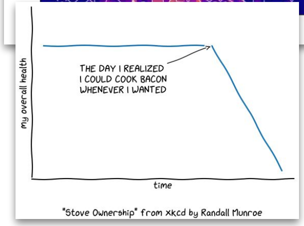
- Plotting is **THE** thing Python **excels** at!
- Matplotlib is inspired by MATLAB and the defacto standard
- A perfect **use for AI**!
  - Can be very tedious
  - Lots of ways to achieve a certain goal
  - Easy to cross-check
- There are millions of guides online:
  - HSF training
  - JetBrains
  - Google










THE DAY I REALIZED I COULD COOK BACON WHENEVER I WANTED
my overall health
time
"Stove Ownership" from xkcd by Randall Munroe

Easter-egg: XKCD

Let’s have a look at some examples

Where alternatives might be better

- Performance-Intensive Applications:
  - Complex projects, like particle physics simulations on large scales
- Parallel & Multi-threaded Computations:
  - Python supports these concepts, but the GIL can be limiting
- Large-Scale Data Analysis & Big Data:
  - Python is memory hungry and huge amounts of data can be limiting
- Low-level system programming and real-time/time critical applications
- **In general: Python can do many things, but the more specific the problem, the worse the solution with Python**
  - E.g. web dev works, but for complex projects, others are definitely better

Wrap-Up Part II

- Python is an awesome language that shines in many many fields
- It's clear and simple philosophy mainly helps beginners getting started and is also at later stages very convenient
- The **rich ecosystem** is extremely valuable
- It excels in prototyping and solving simple (One-shot) problems
- It's main disadvantage is **performance**
  - Can be partly mitigated (see later)
- The more specialized the problem, the more likely you will find a better alternative

- Follow common styles (e.g. in naming) and use meaningful names and always add documentation
  - Your future-You will be thankful!
- Following best practices, such as PEP8, from the beginning is recommended
- **Make use of the rich ecosystem (!)**
- In the beginning, use Python for prototyping and get used to it
- Start simple, keep it simple, and increase complexity only when you realize that you reach boundaries, and never else
  - This e.g. means: only write simple (maybe even atomic) functions

Part III

Language Concepts

Objects

Typing

Memory

(Im)mutable

Dict and Hashable

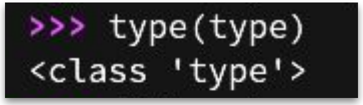
Callable

Errors

OOP

The Main Concepts: Objects

- In Python **EVERYTHING** is an Object!
- Each object is the instance of a class
- However, classes are objects as well...
- How?
- Python defines a **metaclass** called *type*
- From this class, all classes inherit the required members, e.g. __*class*__
- Types (and everything else) are then deduced at runtime

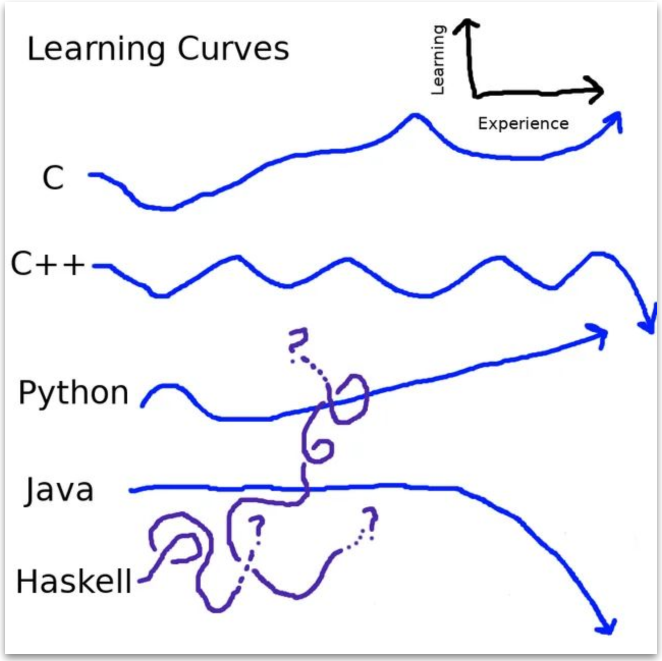


```
>>> type(type)
<class 'type'>
```

EVEYTHING is an object!

The Main Concepts: Objects – But Why?

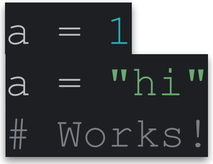
- The idea behind the strong objectification of everything is to abstract away the underlying machine-level complexity
- This makes Python rather **easy to learn** and to use
- On top, this allows to simply use Python straight away and increase code complexity and optimizations with increasing experience



https://www.reddit.com/r/ProgrammerHumor/comments/b55nif/learning_curves/

The Main Concepts: Objects – Dynamic Typing

- Since everything is an object, everything has a type
- In opposite to compiled languages, the type is deduced at runtime
- What does that mean for us?
  - We do not have to care, the interpreter does all the work!
  - Abstracts away all low-level stuff, such as memory management, fct pointer dispatch, c_call/instruction pointer dereferencing, …
  - Downside: This is way more effort than in C (see later)



```
a = 1
a = "hi"
# Works!
```

- Check out the type reference, if you are interested

Interlude: Type Hints

```python
def square(input:int) -> int:
    print(f"Type of input: {type(input)}")
    output = input * input
    print(f"Type of output: {type(output)}")
    return output
```

- Python does not allow static types (unless you use C-types, see later)
- Type hints are not binding for the interpreter, but nevertheless very useful!

```python
x: int = 10
```

- Using a linter (e.g. flake8) helps to write type-*secure* code and avoid errors + improves code quality in general
  - Usage: $ flake8 /path/to/file.py
- Often already integrated in modern IDEs

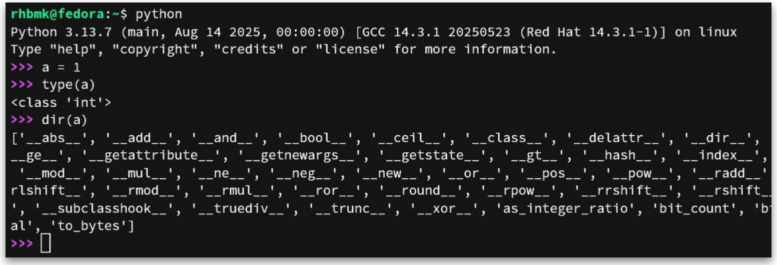
Type hints are **super useful** as they add information on the initiation of the code (e.g. when proper documentation is missing) and can help avoiding bugs and errors with a linter!

Recommendation: Use them straight away from the start

The Main Concepts: Objects – __DUNDER__

Task 1

- Every built in class has several “magic methods” indicated with __*NAME*__
- These are so-called “hooks”, which implement various behaviors of objects, such as:
  - __***add***__ : implements “+”
  - __***str***__ : defines print() output
  - __***repr***__ : fallback for str
- Note: If these are missing (in own classes), Python will use a default, less helpful version like *<__main__.MyClass object at 0x...>*
- Operator Overloading can be realized with *dunders*



```
rhbmk@fedora:~$ python
Python 3.13.7 (main, Aug 14 2025, 00:00:00) [GCC 14.3.1 20250523 (Red Hat 14.3.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 1
>>> type(a)
<class 'int'>
>>> dir(a)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__',
__ge__', '__getattribute__', '__getnewargs__', '__getstate__', '__gt__', '__hash__', '__index__',
 '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__'
rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift_
', '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'as_integer_ratio', 'bit_count', 'b
al', 'to_bytes']
>>> 
```

Dunder methods control core behavior and interactions with Python's built-in operations, while non-dunder methods are for regular functionalities specific to the class itself.

The Main Concepts: Objects – id()

Example: References

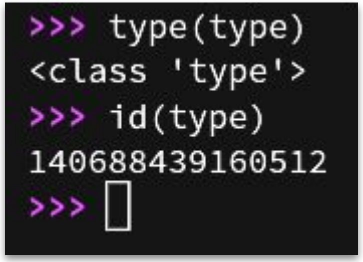
- Everything in Python is an object! Isn't it?

The Main Concepts: Objects – id()

Example: References

- Everything in Python is an object! Isn't it?
  - Except **variables…** Variables are **references to objects** in memory!
- They **do not hold** the actual data themselves
  - When assigning a value to a variable, the variable only **points** to an object in memory
- This is a key concept of Python's **memory management model**
- With the *id()* function, we can for example see that even *type* itself is an instance of the class *type* with a given memory address identifier

(Note: this is not the real hardware address in the CPythong reference implementation!)



```
>>> type(type)
<class 'type'>
>>> id(type)
140688439160512
>>> 
```

The Main Concepts: Objects – Reference Counting

- Python keeps track of all references to an object as part of the internal **memory management**
- When an object is created, Python automatically allocates memory and tracks it as a reference
- When all references are gone (refcount==0), this allows an automatic **Garbage Collection** (GC)
- Hence, we do not need to free memory ourselves, but Python handles it
    - Again a nice feature of the abstraction and very beginner-friendly
- But, it has some twists (see examples)

Keep track of references and copies, if memory can be limiting. Consider deep-copy.
Also important for mutable objects, if you do not want to alter the original data!

The Main Concepts: Objects – Memory

Example: References

- Python is rather memory hungry, but provides multiple ways to keep track:
    - **sys.getsizeof()**: Note that this only gives the immediate memory usage of the object and does not consider referenced objects.
    - **tracemalloc**: Builtin for tracking memory usage over time and comparing memory allocations at different points during execution. This method tracks memory snapshots.
    - **Modules**: memory_profiler, psutil
    - **External tools**: htop, smem, pmap -x <pid>, $ ps -p <pid> -o %mem,vsz,rss, vmstat 1

Use context managers to avoid stale references and safely handle I/O

The Main Concepts: (Im)mutable

Example: (Im)mutable

- Python has two types of builtin data types:
    - **mutable**: dict, list, set
    - **immutable**: int, float, str, frozenset
- Mutables can be changed after creation
- The type has an effect on the references, see example!
- Immutables always require a re-assignment on change
    - -> **more compute intense,** but inherently **thread safe**
- Immutables have a **consistent hash** during their lifetime

The Main Concepts: Dictionaries and Hashables

Example: (Im)mutable

- Dicts in Python are an abstraction of hash maps
- They provide an extremely fast key:value lookup: **O(1)**
  - This even holds for large sizes
- How?
  - A key has to be hashable; each key has to be unique in a dict/set
  - The hash function essentially maps the key to an index in a fixed-size array, the **hash table**
  - **Lookup**: Python simply calculates the hash of the key and checks the corres-ponding memory location for the value

Use dicts, they are nice and fast! Keep in mind that hashes are not unique and be careful in assignments and dict creations.

The Main Concepts: Callable and Callback

Callable

- Callables are simply objects in Python that implement __*call*__
  - This means, they are callable with "()" -> just like functions
- They are the basis for pipelining in Python (stacking function calls by providing functions as callable arguments to other functions)
- They allow **callbacks** and other handy function handlings, like **decorators**
-

Callbacks can be a nice thing, especially in multithreading/asyncio applications. Following the simplicity approach, I do not recommend to use them, if no urgent need.

The Main Concepts: Callable – Decorators

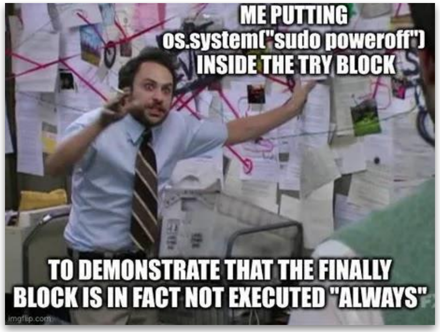
Callable

- A decorator in Python is a function that takes another function and extends or alters its behavior without actually changing the function's code ("decorates" it)
  - -> Decorators in Python are useful whenever you want to add some reusable functionality to functions or methods in a clean and readable way
- They are often used for logging, timing, access control, ...
- Under the hood: It's simply a callable!

Decorators can be really useful!
But in practice (for me at least) I do not find so many application in addition to the ones mentioned...

The Main Concepts: Error Handling

- Error handling is important for good code (and in general a good coding practice)
- In Python, **try…except…finally** implements the error handling
  - finally is always executed!
- **Explicitly** catching error types is considered **good practice**


ME PUTTING
os.system("sudo poweroff")
INSIDE THE TRY BLOCK
TO DEMONSTRATE THAT THE FINALLY
BLOCK IS IN FACT NOT EXECUTED "ALWAYS"
imgflip.com

Errors are very important in Python, see Zen of Python and builtins!
A proper handling together with tests helps a lot reducing bug potential and increasing code quality!

Should I use OOP in Python?

Classes can be wild

- I'd say: Of course, if you **cannot avoid** it.
- It is a powerful part of the language (heavily building on objects…)
- BUT: It can create an overhead in complexity and performance
  - Also for other languages! (C is often faster than C++)
- I've used it, but for many tasks, it is not required, nor recommended, as simpler solutions are to be preferred (Remember: Zen of Python)
- My recommendation for beginners:
  - **Avoid it**. Only well planned, mid- to long-term projects profit from it

From my personal experience: OOP makes sense, when you move from a One-shot **script** to an **application** with a well-design concept, different source files, etc. In such a case, it is clearly recommended, as it increases maintainability and allows more complex and optimized projects.