General Introduction to Important Python Features

FTAG algo tutorial on good code practices, 14.04.2022

Manuel Guth

with Material from GRK python workshop (in cooperation with Frank Sauerburger) and RODEM good practices mini-workshop

Overview

- General good practices
- New Features in Python 3
- Generators
- Type hinting / Type declaration
- Logging
- argparse
- What not to do
- Debugger
- Code formatting & Linting

Auxiliary Material (for which we don't have enough time)

- Object-Oriented Programming)

reusing some material from https://indico.cern.ch/event/846501

General good practices

- Comment your Code
- Add doc strings

New Features in Python 3

New Features in Python 3

- Python 2 is deprecated since beginning of 2020
- Python 3 already has 10 minor releases (3.xx)
- For all changes have a look at What's New in Python
- Cheat Sheet: Writing Python 2-3 compatible code

are your libraries ready for python 3.10? have a look

f-Strings

f-Strings

String formatting before Python 3.6

In [ ]:
```
import math
ftag = 2_022
where = "online"
```

f-Strings

String formatting before Python 3.6

In [ ]:

```
import math
ftag = 2_022
where = "online"
```

In [ ]:

```
message = "Welcome to the FTAG Algo {} Good practice tutorial {}!\nWe c
```

f-Strings

String formatting before Python 3.6

In [ ]:

```
import math
ftag = 2_022
where = "online"
```

In [ ]:

```
message = "Welcome to the FTAG Algo {} Good practice tutorial {}!\nWe c
```

In [ ]:

```
print(message)
```

f-Strings

String formatting before Python 3.6

In [ ]:
```
import math
ftag = 2_022
where = "online"
```

In [ ]:
```
message = "Welcome to the FTAG Algo {} Good practice tutorial {}!\nWe c
```

In [ ]:
```
print(message)
```

String formatting with f-String

In [ ]:
```
message_f = f"Welcome to the FTAG Algo {ftag} Python mini workshop {whe
```

f-Strings

String formatting before Python 3.6

In [ ]:
```
import math
ftag = 2_022
where = "online"
```

In [ ]:
```
message = "Welcome to the FTAG Algo {} Good practice tutorial {}!\nWe c
```

In [ ]:
```
print(message)
```

String formatting with f-String

In [ ]:
```
message_f = f"Welcome to the FTAG Algo {ftag} Python mini workshop {whe
```

In [ ]:
```
print(message_f)
```

True Division

Python 2

3/4 returned 0

True Division

Python 2

3/4 returned 0

Python 3

In [ ]:
```
3/4
```

In python 3 the operator `/` does not loose fractions

True Division

Python 2

3/4 returned 0

Python 3

In [ ]:
```
3/4
```

In python 3 the operator `/` does not loose fractions

Integer division has its own operator

In [ ]:
```
3//4
```

Readability of Numbers

To make large number better readable, you can use a `_`

In [ ]:
```
6728339
```

In [ ]:
```
6_728_339
```

Dictionary operators

New Merge (|) and update (|=) operators for dictionaries

Dictionary operators

New Merge (|) and update (|=) operators for dictionaries

In [ ]:
```
dict1 = {"key1": "CERN", "key2": "DESY"}
dict2 = {"key2": "CH", "key3": "DE"}
```

In [ ]:
```
dict1 | dict2
```

In [ ]:
```
dict2 | dict1
```

In [ ]:
```
dict2 |= dict1
```

In [ ]:
```
dict2
```

- available since python 3.9

Parenthesised context managers

In [ ]:

```python
with (
    CtxManager1() as example1,
    CtxManager2() as example2,
    CtxManager3() as example3,
):
    ...
```

In [ ]:

Structural Pattern Matching

In [ ]:

```python
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

In [ ]:

```python
http_error(418)
```

Pairwise function itertools

In [ ]:

```
from itertools import pairwise
words = ["good", "morning", "routine"]
for w1, w2 in pairwise(words):
    print(w1, w2)
```

- available since python 3.10
- useful e.g. when looping over indices for batches

Pydash

In [ ]:

```
from pydash import flatten_deep, map_, omit
```

Pydash

In [ ]:

```
from pydash import flatten_deep, map_, omit
```

In [ ]:

```
# flatten a nested list
flatten_deep([1, 2, 3, [4, 5, 6, [7, 8, 9]], [2]])
```

Pydash

In [ ]:

```
from pydash import flatten_deep, map_, omit
```

In [ ]:

```
# flatten a nested list
flatten_deep([1, 2, 3, [4, 5, 6, [7, 8, 9]], [2]])
```

In [ ]:

```
# filter specific values from a list of dictionaries
map_([{"letter": "alpha", "position": 1}, {"letter": "delta", "positior
```

Pydash

In [ ]:
```
from pydash import flatten_deep, map_, omit
```

In [ ]:
```
# flatten a nested list
flatten_deep([1, 2, 3, [4, 5, 6, [7, 8, 9]], [2]])
```

In [ ]:
```
# filter specific values from a list of dictionaries
map_([{"letter": "alpha", "position": 1}, {"letter": "delta", "position
```

In [ ]:
```
# remove key from dictionary
omit({"letter": "eta", "position": 7}, "position")
```

Useful package for list, dictionary handling

Generators

In [ ]:
```
def squares(end):
    """
    Returns the squares of 0 up to (not including) the given end.
    >>> squares(3)
    [0, 1, 4]
    """
    out = []
    for i in range(end):
        out.append(i * i)
    return out
```

In [ ]:
```
squares(3)
```

This is a typical pattern:

1. Create empty list
2. Append items in loop
3. Return final list

Problematic when dealing with huge lists

In [ ]:
```
small_list = squares(10)   # Returns list of 10 items
sum(small_list)
```

In [ ]:
```
large_list = squares(1000_000)  # Returns a list with 1 million items
                                # Calling it with 1 billion exhausts my
sum(large_list)
```

In this example

- Don't need random access to items: `large_list[100]`
- Need only to iterate over list once

Solution: Generators

In [ ]:

```
def squares(end):
    """
    Returns the squares of 0 up to (not including) the given end.
    >>> squares(3)
    [0, 1, 4]
    """
    # Old implemenation:
    # out = []
    # for i in range(end):
    #     out.append(i * i)
    # return out
    for i in range(end):
        yield i * i  # yield one item at a time
```

In [ ]:

```
squares(3)
```

In [ ]:

```
list(squares(3))
```

In [ ]:

```
sum(squares(1000_000))  # Computes one item at a time
# Works even with 1 billion, takes ~2min
```

Type hinting / Type declaration

In [ ]:

```python
def multiply_values(val1, val2):
    """Multiplies two floats and returns result."""
    return f"Result: {val1 * val2}"
```

Type hinting / Type declaration

In [ ]:

```python
def multiply_values(val1, val2):
    """Multiplies two floats and returns result."""
    return f"Result: {val1 * val2}"
```

In [ ]:

```python
multiply_values(5.4, 1.2)
```

In [ ]:

```python
multiply_values(5, 2)
```

In [ ]:

```python
multiply_values(True, False)
```

Type hinting / Type declaration

In [ ]:
```
def multiply_values(val1, val2):
    """Multiplies two floats and returns result."""
    return f"Result: {val1 * val2}"
```

In [ ]:
```
multiply_values(5.4, 1.2)
```

In [ ]:
```
multiply_values(5, 2)
```

In [ ]:
```
multiply_values(True, False)
```

Common case!

- Function intended to be used with floats
- Python doesn't forbid other types

How to avoid that

How to avoid that

Type hinting helps to remind yourself and other developers about your intentions

- Hinted types of arguments
- Hinted return type

How to avoid that

Type hinting helps to remind yourself and other developers about your intentions

- Hinted types of arguments
- Hinted return type

In [ ]:

```
def multiply_values(val1: float, val2: float) -> str:
    """Multiplies two floats and returns result."""
    return f"Result: {val1 * val2}"
```

How to avoid that

Type hinting helps to remind yourself and other developers about your intentions

- Hinted types of arguments
- Hinted return type

In [ ]:
```
def multiply_values(val1: float, val2: float) -> str:
    """Multiplies two floats and returns result."""
    return f"Result: {val1 * val2}"
```

Can ask for the type hints at run time:

In [ ]:
```
from typing import get_type_hints
```

In [ ]:
```
get_type_hints(multiply_values)
```

A few reminders

Type hints are just *hints*, they do not declare types. Can still do this:

In [ ]:
```
multiply_values(True, False)
```

A few reminders

Type hints are just *hints*, they do not declare types. Can still do this:

In [ ]:
```
multiply_values(True, False)
```

In [ ]:
```
get_type_hints(multiply_values)
```

A few reminders

Type hints are just *hints*, they do not declare types. Can still do this:

In [ ]:
```
multiply_values(True, False)
```

In [ ]:
```
get_type_hints(multiply_values)
```

> *Python will remain a dynamically typed language, and the authors have no desire to ever make type hints mandatory, even by convention.*

Logging

Logging

… defines functions and classes which implement a flexible event logging system for applications and libraries.

- Track the status of software at runtime
- Can be output, stored to a file, etc.
- Can have different severity/importance levels
- Can have custom output format

Logging levels

- `DEBUG` – detailed information, only for problem diagnosis
- `INFO` – conformative, "working as expected"
- `WARNING` – something unexpected happened, maybe a problem in the near future, but: still working as expected
- `ERROR` – more serious problem, some operation not executed
- `CRITICAL` – serious error, program itself might be compromised

Loggers, Handlers, Formatters

Loggers, Handlers, Formatters

- Loggers: to expose the interface that applications use
- Handlers: to send the logs to the appropriate destination
- Formatters: to specify the log layout in the final output

Loggers, Handlers, Formatters

- Loggers: to expose the interface that applications use
- Handlers: to send the logs to the appropriate destination
- Formatters: to specify the log layout in the final output

In [ ]:
```python
import logging
```

Loggers, Handlers, Formatters

- Loggers: to expose the interface that applications use
- Handlers: to send the logs to the appropriate destination
- Formatters: to specify the log layout in the final output

In [ ]:
```
import logging
```

In [ ]:
```
logger = logging.getLogger()
logger.setLevel("INFO")
```

In [ ]:
```
handler = logging.StreamHandler()
```

Loggers, Handlers, Formatters

- Loggers: to expose the interface that applications use
- Handlers: to send the logs to the appropriate destination
- Formatters: to specify the log layout in the final output

In [ ]:
```
import logging
```

In [ ]:
```
logger = logging.getLogger()
logger.setLevel("INFO")
```

In [ ]:
```
handler = logging.StreamHandler()
```

In [ ]:
```
formatter = logging.Formatter(
    "%(funcName)s()  %(levelname)7s  %(message)s",
    '%H:%M:%S'
)
handler.setFormatter(formatter)
```

In [ ]:
```
logger.addHandler(handler)
```

Simple example

In [ ]:
```python
def floor(var: float) -> int:
    """Floors a float."""
    logger.info(f"called with argument var={var}.")
    if type(var) not in [float, int]:
        logger.error(
            f"called with var={var} which is neither float nor int."
            " Returned 'None' as I don't know what to do here."
        )
        return None
    elif type(var) is not float:
        logger.warning(f"called with var={var} which is not a float.")

    return int(var)
```

Simple example

In [ ]:

```python
def floor(var: float) -> int:
    """Floors a float."""
    logger.info(f"called with argument var={var}.")
    if type(var) not in [float, int]:
        logger.error(
            f"called with var={var} which is neither float nor int."
            " Returned 'None' as I don't know what to do here."
        )
        return None
    elif type(var) is not float:
        logger.warning(f"called with var={var} which is not a float.")

    return int(var)
```

In [ ]:

```python
floor(3.7)
```

In [ ]:

```python
floor(3)
```

In [ ]:

```python
floor("3")
```

More things to be done with loggers – some ideas

More things to be done with loggers – some ideas

- Multiple handlers, e.g. to:
    - send warning/error/fatal to std output
    - send info/warning/error/fatal to a log file
    - ...
- Same or different formats for multiple handlers
- Make use of a command-line argument `--debug` to:
    - print everything down to debug level to std output
    - use a different formatter that prints more info (e.g. module name + line number)

Command-line options - `argparse`

Command-line parsing module in the Python standard library

Command-line options - `argparse`

Command-line parsing module in the Python standard library

All sorts of configurations possible:

- Positional / Keyword
- Default values
- Keywords can be mandatory or optional
- Help messages

Command-line options - `argparse`

Command-line parsing module in the Python standard library

All sorts of configurations possible:

- Positional / Keyword
- Default values
- Keywords can be mandatory or optional
- Help messages

In [ ]:

```
from argparse import ArgumentParser
```

Command-line options - `argparse`

Command-line parsing module in the Python standard library

All sorts of configurations possible:

- Positional / Keyword
- Default values
- Keywords can be mandatory or optional
- Help messages

In [ ]:
```
from argparse import ArgumentParser
```

In [ ]:
```
parser = ArgumentParser()
```

How to add arguments

How to add arguments

In [ ]:

```
parser.add_argument("number", type=float) # positional argument with ty
```

How to add arguments

In [ ]:
```
parser.add_argument("number", type=float) # positional argument with ty
```

In [ ]:
```
parser.add_argument(
    '-e',             # short-hand
    '--exponent',     # full name
    default=2,        # default value
    type=int,         # int type
)
```

How to add arguments

```
In [ ]: parser.add_argument("number", type=float) # positional argument with ty
```

```
In [ ]: parser.add_argument(
            '-e',               # short-hand
            '--exponent',       # full name
            default=2,          # default value
            type=int,           # int type
        )
```

```
In [ ]: parser.add_argument(
            "-v",                                 # short-hand
            "--verbose",                          # full name
            help="increase output verbosity",     # help message
            action="store_true",                  # true/false
        )
```

What NOT to do

Thinks you should avoid with python

Misusing default arguments in functions

you can define default values in a function

Misusing default arguments in functions

you can define default values in a function

In [ ]:

```
def ftag_append(ftag_list=[]):  # ftag_list is optional with the defaul
    ftag_list.append("algo") # this line can cause problems!
    return ftag_list
```

Misusing default arguments in functions

you can define default values in a function

In [ ]:
```
def ftag_append(ftag_list=[]):  # ftag_list is optional with the defaul
    ftag_list.append("algo") # this line can cause problems!
    return ftag_list
```

In [ ]:
```
ftag_append()
```

Misusing default arguments in functions

you can define default values in a function

In [ ]:
```
def ftag_append(ftag_list=[]):  # ftag_list is optional with the defaul
    ftag_list.append("algo") # this line can cause problems!
    return ftag_list
```

In [ ]:
```
ftag_append()
```

Possible way out of it

In [ ]:
```
def ftag_append(ftag_list=None):  # setting default value to None
    if ftag_list is None:
        ftag_list = []
    ftag_list.append("algo")
    return ftag_list
```

Misusing default arguments in functions

you can define default values in a function

In [ ]:
```
def ftag_append(ftag_list=[]):  # ftag_list is optional with the defaul
    ftag_list.append("algo") # this line can cause problems!
    return ftag_list
```

In [ ]:
```
ftag_append()
```

Possible way out of it

In [ ]:
```
def ftag_append(ftag_list=None):  # setting default value to None
    if ftag_list is None:
        ftag_list = []
    ftag_list.append("algo")
    return ftag_list
```

In [ ]:
```
ftag_append()
```

Import Mistakes

Wildcard Import

Import Mistakes

Wildcard Import

In [ ]:
```
from numpy import *
```

- Can cause name clashing
- Unnecessary import of unneeded functionalities

Import Mistakes

Wildcard Import

In [ ]:
```
from numpy import *
```

- Can cause name clashing
- Unnecessary import of unneeded functionalities

with python 3 e.g. ROOT does not allow wildcard import anymore

```
from ROOT import *
```

Import Mistakes

Name conflicts with other libraries

Import Mistakes

Name conflicts with other libraries

email is a python standard library

```
from email.message import EmailMessage
```

Import Mistakes

Name conflicts with other libraries

email is a python standard library

```
from email.message import EmailMessage
```

```
%%writefile email.py
def GetMail():
    return "grk@physik.uni-freiburg.de"
```

Import Mistakes

Name conflicts with other libraries

email is a python standard library

```
from email.message import EmailMessage
```

```
%%writefile email.py
def GetMail():
    return "grk@physik.uni-freiburg.de"
```

```
import email
email.GetMail()
```

Opening files

Often used to open files

```
file = open("test.txt", "w")
.
.
.
file.close()
```

This synthax can cause issues e.g. if there is an exception raised before `file.close()`

Opening files

Often used to open files

```
file = open("test.txt", "w")
.
.
.
file.close()
```

This synthax can cause issues e.g. if there is an exception raised before `file.close()`

Saver way to open files

```
with open("test.txt", "w") as file:
    .
    .
    .
```

Mutable assignment errors - Dictionaries

We have a dictionary a

In [ ]:

```
a = {'1': "one", '2': 'two'}
```

Mutable assignment errors - Dictionaries

We have a dictionary a

In [ ]:
```
a = {'1': "one", '2': 'two'}
```

Now we want to have the same dict again but leaving the previous one intact

Mutable assignment errors - Dictionaries

We have a dictionary a

In [ ]:
```
a = {'1': "one", '2': 'two'}
```

Now we want to have the same dict again but leaving the previous one intact

In [ ]:
```
b = a
```

Mutable assignment errors - Dictionaries

We have a dictionary a

In [ ]:
```
a = {'1': "one", '2': 'two'}
```

Now we want to have the same dict again but leaving the previous one intact

In [ ]:
```
b = a
```

In [ ]:
```
b
```

Mutable assignment errors - Dictionaries

We have a dictionary a

In [ ]:
```
a = {'1': "one", '2': 'two'}
```

Now we want to have the same dict again but leaving the previous one intact

In [ ]:
```
b = a
```

In [ ]:
```
b
```

In [ ]:
```
b['3'] = "three"
```

Mutable assignment errors - Dictionaries

We have a dictionary a

In [ ]:
```
a = {'1': "one", '2': 'two'}
```

Now we want to have the same dict again but leaving the previous one intact

In [ ]:
```
b = a
```

In [ ]:
```
b
```

In [ ]:
```
b['3'] = "three"
```

In [ ]:
```
a
```

Mutable assignment errors - Dictionaries

What happened?

Here b is a pointer -> reference to a.

The same thing is happening for lists.

Mutable assignment errors - Dictionaries

What happened?

Here b is a pointer -> reference to a.

The same thing is happening for lists.

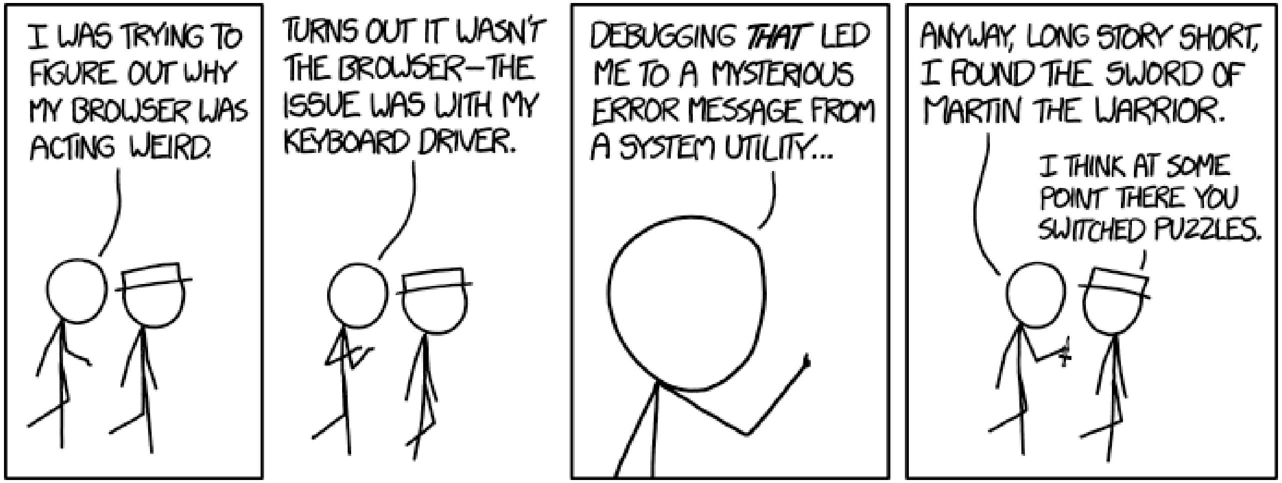
Possible way out:

In [ ]:
```
# for dicts
b = a.copy()
# for lists
l = list(a.keys())
cp = l[:]
```

Debugger PDB

Your program crashes or doesn't do what it should?

Debugging can be challenging



Example

In [ ]:

```python
from myproject import read_config, compute_all_results

config = read_config()
# ...
results = compute_all_results(config)  # lengthy computation
# ...
for result in results:
    if result == "tt":
        print("We have the answer!")
        break
else:
    print("This should not happen.")
```

Debugging with `print()`

Add single print, rerun **whole** program

In [ ]:

```
config = read_config()
# ...
results = compute_all_results(config)  # lengthy computation
# ...
print(results)  # Inspect the list of results
for result in results:
    if result == "tt":
        print("We have the answer!")
        break
else:
    print("This should not happen.")
```

Debugging with `print()`

Add single print, rerun **whole** program

In [ ]:
```python
config = read_config()
# ...
results = compute_all_results(config)  # lengthy computation
# ...
print(results)  # Inspect the list of results
for result in results:
    if result == "tt":
        print("We have the answer!")
        break
else:
    print("This should not happen.")
```

- `tt` in results
- Why not detected in loop?

Debugging with `print()`

Add another print, rerun **whole** program **again**

In [ ]:

```
config = read_config()
# ...
results = compute_all_results(config)  # lengthy computation
# ...
print(results)  # Inspect the list of results
for result in results:
    print(result)
    if result == "tt":
        print("We have the answer!")
        break
else:
    print("This should not happen.")
```

Better: Using debugger

Insert `breakpoint()` (or `import pdb; pdb.set_trace()` before Python 3.7) and rerun whole program

In [ ]:

```
config = read_config()
# ...
results = compute_all_results(config)  # lengthy computation
# ...
import pdb; pdb.set_trace()  # This works also before 3.7
for result in results:
    if result == "tt":
        print("We have the answer!")
        break
else:
    print("This should not happen.")
```

Better: Using debugger

- Trigger debugger
  - Add `breakpoint()` or `import pdb; pdb.set_trace()`
  - Run `python -m pdb your_program.py`
- Command summary
  - `b [FILE:]LINE` adds a new **b**earkpoint
  - `c` **c**ontinue to next breakpoint
  - `n` run **n**ext statement
  - `s` **s**tep into method call
  - `u` move one level up (reverts `s`)
  - `cl [N]` clear breakpoints or breakpoint `N`
  - `q` **q**uit
  - `h` **h**elp

Exercise:

Investigate the example below:

In [ ]:
```
cities = set(["London", "Paris", "Bern"])  # Unordered collection

def get_new_cities():
    new_cities = []
    new_cities.append("Oslo")
    new_cities.append("Praque")
    return set(new_cities)

cities.union(get_new_cities())

print(cities)  # Does not include Oslo, Praque!
```

Code formatting & Linting

Code formatter = runs over your code and applies styling changes

Linter = scans the code to flag:

- Programming errors / invalid syntax
- Suspicious constructs ("code that smells")
- Stylistic errors (enforces common style within a team)

The combination of the two is extremely powerful!

Linter example

The slightly modified example of the cities.

```python
# debug_exercise.py
cities = ["London", "Paris", "Bern"]

def get_nordic_cities():
    cities = []
    cities.append("Oslo")
    cities.append("Stockholm")
    return cities

nordic_cities = get_nordic_cities()

print(cities)  # Still contains London, Paris, Bern
```

Linter example

```
$ python -m pylint debug_exercise.py
example.py:1:0: C0114: Missing module docstring (missing-module-
docstring)
example.py:6:4: W0621: Redefining name 'cities' from outer scope
(line 3) (redefined-outer-name)
example.py:5:0: C0116: Missing function or method docstring (missing-
function-docstring)

------------------------------------------------------------------
Your code has been rated at 6.25/10 (previous run: 6.25/10, +0.00)
```

- `cities` redefined within the function
- In this example the redefinition might be obvious and not a problem
- But what if the code is much more complex? Shadowing is dangerous!
- Linter would have given a hint of the problem already

What to take away

- Code formatter, e.g. `black`, to have uniform code style
- `pylint`, `flake8`, ... + other style checkers to cross-check syntax, constructs etc

The best is the combination of both! Ideal for pre-commit hooks & CI/CD:

- Pre-commit hooks – no "broken" commits:
    - code formatter
    - style checker / linter
    - other safety nets, e.g. yaml syntax checker
- Continuous integration: linter + all actual code tests

Auxiliary Material

several concepts of python we couldn't cover in the tutorial

Print Function

Print Function

In [ ]:

```
print("Hello world!")
```

Print Function

In [ ]:

```
print("Hello world!")
```

In [ ]:

```
print("Hello", "world", sep="-")
```

Print Function

In [ ]:
```
print("Hello world!")
```

In [ ]:
```
print("Hello", "world", sep="-")
```

In [ ]:
```
print('home', 'user', 'documents', sep='/')
```

Print Function

In [ ]:

```
print('Mercury', 'Venus', 'Earth', sep=', ', end=", ")
print('Mars', 'Jupiter', 'Saturn', sep=', ', end=', ')
print('Uranus', 'Neptune', 'Pluto', sep=', ')
```

Print Function

In [ ]:

```
print('Mercury', 'Venus', 'Earth', sep=', ', end=", ")
print('Mars', 'Jupiter', 'Saturn', sep=', ', end=', ')
print('Uranus', 'Neptune', 'Pluto', sep=', ')
```

Writing to file

In [ ]:

```
!cat file.txt
```

In [ ]:

```
with open('file.txt', mode='w') as file_object:
    print('hello world', file=file_object)
```

Dataclass

in Python 3 dataclasses were introduced, for more details have a look here

The dataclass will handle the `__init__` etc

In [ ]:
```
from dataclasses import dataclass

@dataclass
class InventoryItem:
    """Class for keeping track of an item in inventory."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

What is Object-Oriented Programming (OOP)

- You've used it already:

```python
"Hello World".lower()
```

  The string `"Hello World"` is an object of `str` class.

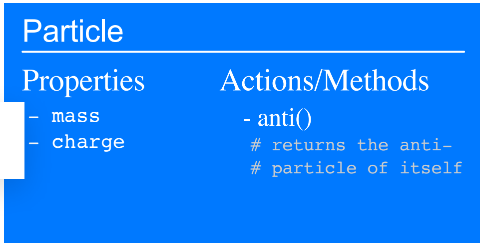
- Class is a *blueprint* to create instances, called *objects*
- Combines data and functions
- Example: Particles in an experiment

Particle

Properties	Actions/Methods

`- mass`
`- charge`
	- anti()
	`# returns the anti-`
	`# particle of itself`

What is Object-Oriented Programming (OOP)



- You've used it already:

```
"Hello World".lower()
```

The string `"Hello World"` is an object of `str` class.

- Class is a *blueprint* to create instances, called *objects*
- Combines data and functions
- Example: Particles in an experiment

In [ ]:

```
class Particle:
    def __init__(self, mass, charge):
        self.mass = mass
        self.charge = charge
```

What is Object-Oriented Programming (OOP)

- You've used it already:

```python
"Hello World".lower()
```

The string `"Hello World"` is an object of `str` class.

Particle

Properties	Actions/Methods
- mass - charge	- anti() # returns the anti- # particle of itself

- Class is a *blueprint* to create instances, called *objects*
- Combines data and functions
- Example: Particles in an experiment

In [ ]:

```python
class Particle:
    def __init__(self, mass, charge):
        self.mass = mass
        self.charge = charge
```

In [ ]:

```python
bert = Particle(125, 0)
bert.mass
```

In [ ]:

```python
class Particle:
    def __init__(self, mass, charge):
        # __init__() is called when new object is created.
        # First argument (self) is the new object
        self.mass = mass
        self.charge = charge

    def anti(self):
        # First argument is the object on which anti() is called

        # Create new particle with same mass and
        # opposite charge
        return Particle(self.mass, -self.charge)
```

In [ ]:

```python
bert = Particle(1.777, -1)
ernie = bert.anti()
ernie.charge
```

In [ ]:

```python
ernie.mass
```

In [ ]:

```python
bert.charge  # Original particle not changed
```

In [ ]:

```
class Particle:
    def __init__(self, mass, charge):
        # __init__() is called when new object is created.
        # First argument (self) is the new object
        self.mass = mass
        self.charge = charge

    def anti(self):
        # First argument is the object on which anti() is called

        # Create new particle with same mass and
        # opposite charge
        return Particle(self.mass, -self.charge)

    def flip_charge(self):
        # Change the charge of the particle itself (instead of creating

        self.charge *= -1
```

In [ ]:

```
bert = Particle(1.777, -1)
bert.charge
```

In [ ]:

```
bert.flip_charge()  # Changes the original particle
bert.charge
```

Inheritance

Inheritance

- Sub-classes extend parent classes
- Share functionality implemented in parent classes
- Terminology: parent class = "base class"; sub-class = "derived class"
- Inheritance models = "**is a**"-type relationships
  - A `Fermion` **is a** `Particle`
  - A `Particle` is not necessariliy a `Fermion`
- Example: Include sub-classes `Fermion` and `Boson`

Inheritance

- Sub-classes extend parent classes
- Share functionality implemented in parent classes
- Terminology: parent class = "base class"; sub-class = "derived class"
- Inheritance models = "**is a**"-type relationships
    - A `Fermion` **is a** `Particle`
    - A `Particle` is not necessariliy a `Fermion`
- Example: Include sub-classes `Fermion` and `Boson`

In [ ]:
```
class Boson(Particle):
    def interact_with_higgs(self, factor=1.5):
        # Bosons can increase their mass by interacting with the Higgs
        self.mass *= factor

class Fermion(Particle):
    def __init__(self, mass, charge, generation):
        super().__init__(mass, charge)  # Create a regular particle
        self.generation = generation
```

In [ ]:

```
tau = Fermion(1.777, -1, 3)
tau.generation
```

In [ ]:

```
Z = Boson(60.78, 0)
Z.mass
```

In [ ]:

```
Z.interact_with_higgs()
Z.mass
```

In [ ]:

```
Z.generation  # Z is a Boson which do not come in generations
```

In [ ]:

```
tau.interact_with_higgs()
```

Other interesting things about OOP

- Methods `__str__` and `__repr__` can be overridden
    - Reminder: `__repr__` = unambiguous representation of an object
    - Reminder: `__str__` = "pretty" printable representation (defaults to `__repr__`)
- Operators can be overriden: `ernie + bert`
- Polymorphism: methods with different implementations sub-classes, e.g.
    - `Fermion.susy()` returns a Boson
    - `Boson.susy()` returns a Fermion