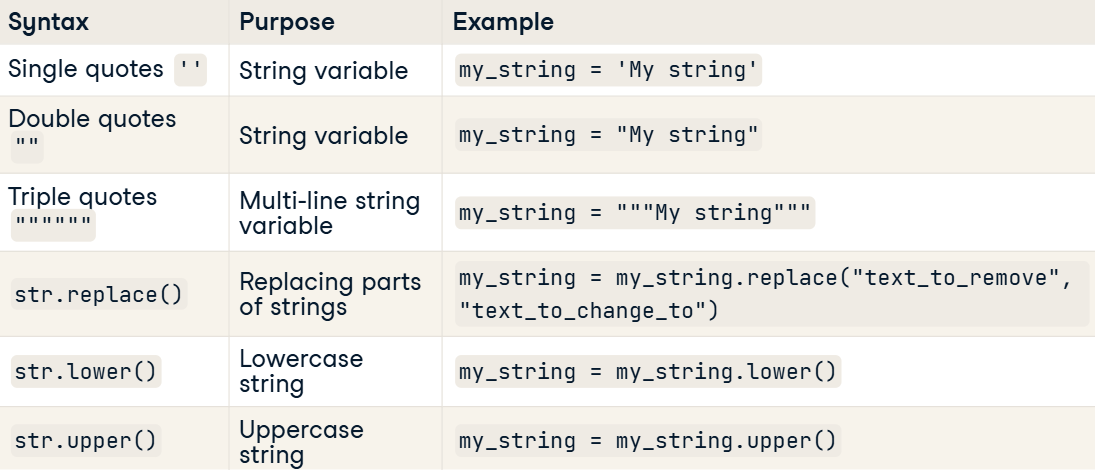
# Python

## Introduction to Python for Developers

### Working with Data Types

#### Working with strings

* Method – a function that is only available to a specific data type.
  + Cheat sheet:

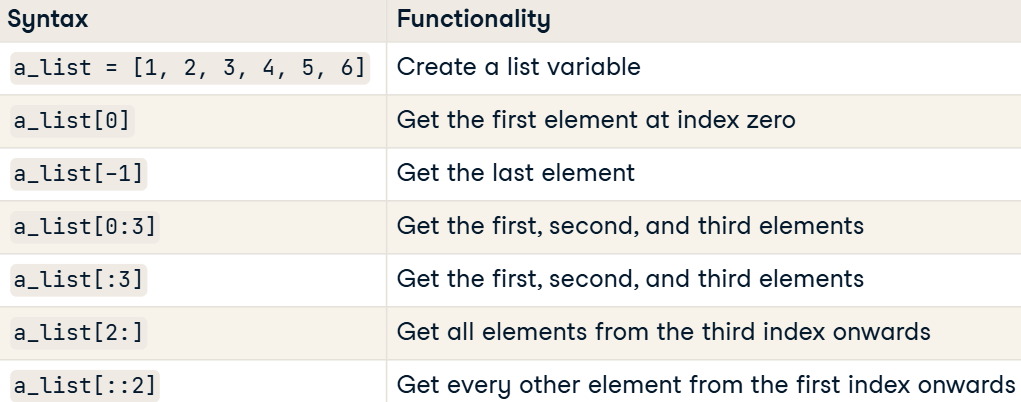


#### Working with lists

* What is it and how it works:

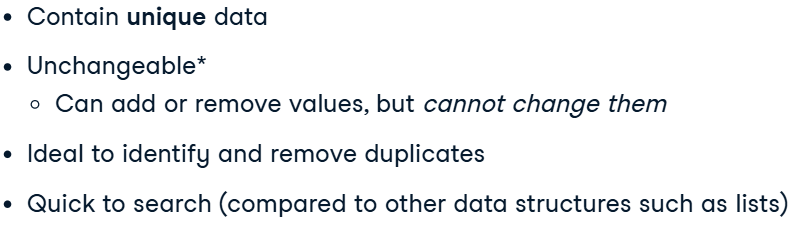


* + Cheat sheet:

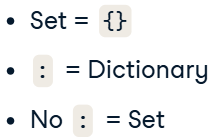


#### Sets and tuples

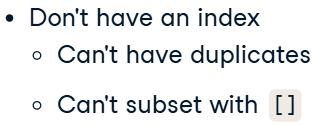
* What are **Sets**:



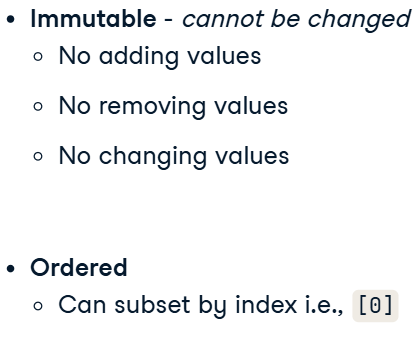
* + Creating sets:



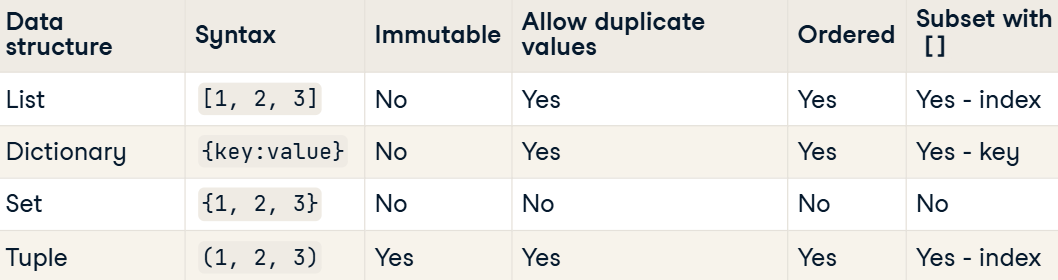
* + Limitations:



* What are **Tuples**:



#### Data structure summary



### Control flow and loops

#### Conditional statements and operators

* If, Elif, Else:

if num\_beds < min\_num\_beds:

  print('Insufficient bedrooms')

elif sq\_foot <= min\_sq\_foot:

  print("Too small")

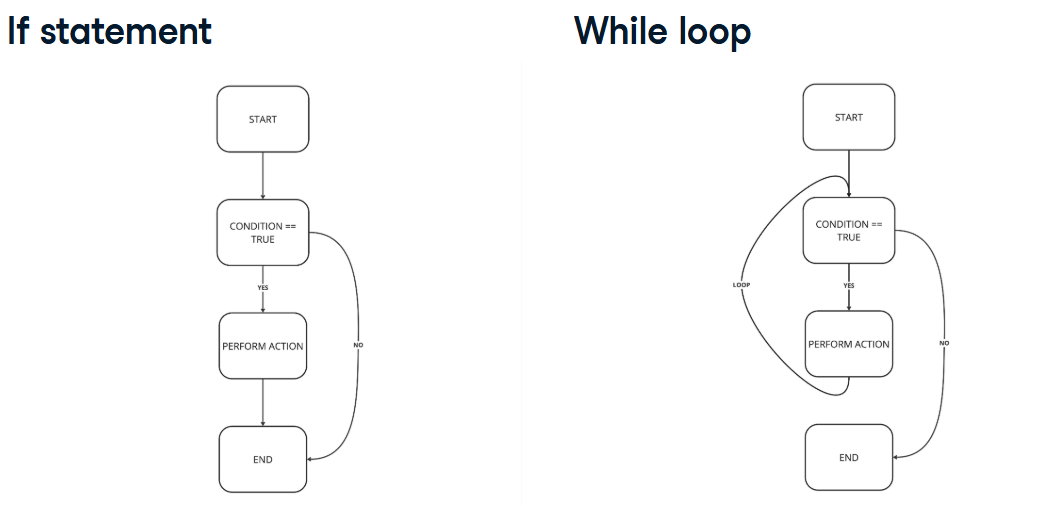
elif rent > max\_rent:

  print("Too expensive")

else:

  print("This looks promising!")

* If VS While:

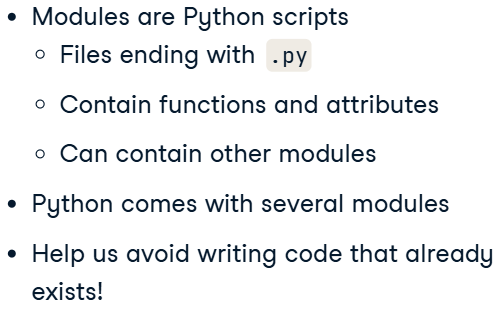


## Intermediate Python for Developers

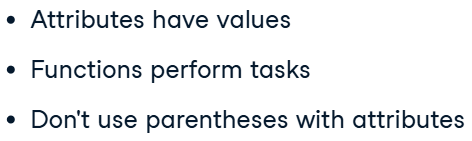
### The Python Ecosystem

#### Modules

* What is module:



* + Attributes:



* Working with directory:



#### Packages

* Package (library) – is a collection of modules.

### Working with functions

#### Docstrings

* Code:

def convert\_data\_structure(data, data\_type="list"):

  """

  Convert a data structure to a list, tuple, or set.

  Args:

    data (list, tuple, or set): A data structure to be converted.

    data\_type (str): String representing the type of structure to convert data to.

  Returns:

    data (list, tuple, or set): Converted data structure.

  """

  if data\_type == "tuple":

    data = tuple(data)

  elif data\_type == "set":

    data = set(data)

  else:

    data = list(data)

  return data

print(help(convert\_data\_structure))

### Lambda functions and error-handling

#### Lambda functions

* Multiple parameters in function:

print((lambda x: x \* 1.2)(sale\_price))

#### Error handling

* Try-Except vs Raise:

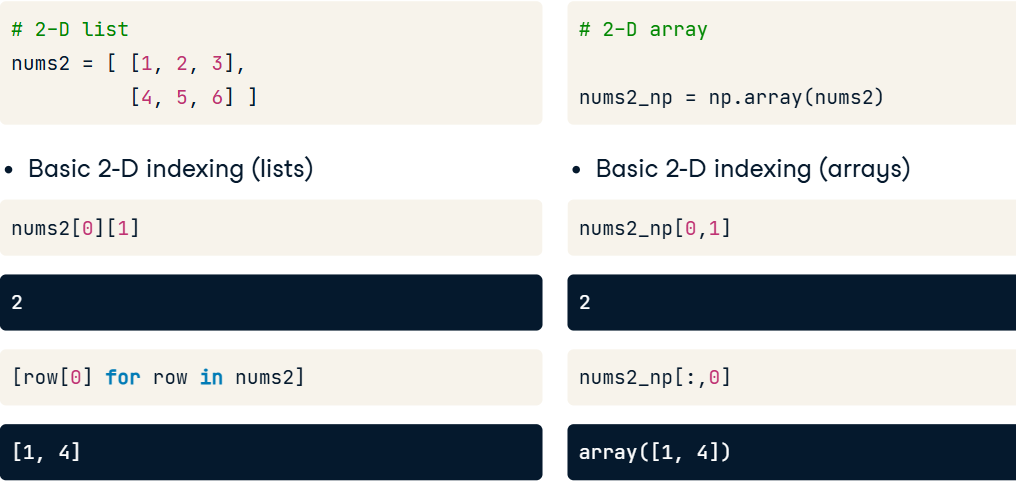


## Writing Efficient Python Code

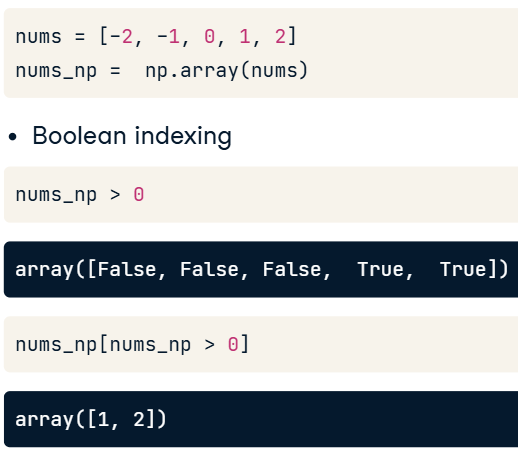
### Foundations for efficiencies

#### The power of NumPy arrays

* Indexing:



* Boolean indexing:



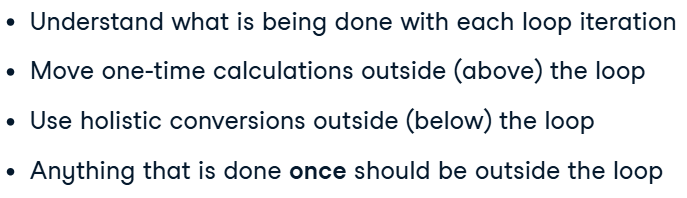
### Gaining efficiencies

#### Set theory

* Better use build-in funcs, brackets, than direct words.
* The fastest way to find desired elements.

#### Writing better loops

* Better looping:



* Code:

possible\_pairs = [\*combinations(pokemon\_types, 2)]

enumerated\_tuples = [] ##1, 4

for i, pair in enumerate(possible\_pairs, start=1):

    enumerated\_pair\_tuple = (i, ) + pair

    enumerated\_tuples.append(enumerated\_pair\_tuple)

enumerated\_pairs = [\*map(list, enumerated\_tuples)] ##1, 3

print(enumerated\_pairs)

### Basic pandas optimizations

#### DataFrame iteration

* **Iterrows**:

run\_diffs = []

for i,row in giants\_df.iterrows():

    runs\_scored = row['RS']

    runs\_allowed = row['RA']

    run\_diff = calc\_run\_diff(runs\_scored, runs\_allowed)

    run\_diffs.append(run\_diff)

giants\_df['RD'] = run\_diffs

print(giants\_df)

* **Iterrtuple** (more efficient, than iterrows, due to using another type of storing rows):

run\_diffs = []

for row in yankees\_df.itertuples():

    runs\_scored = row.RS

    runs\_allowed = row.RA

    run\_diff = calc\_run\_diff(runs\_scored, runs\_allowed)

    run\_diffs.append(run\_diff)

yankees\_df['RD'] = run\_diffs

print(yankees\_df)

* **Apply**:

win\_percs = dbacks\_df.apply(lambda row: calc\_win\_perc(row['W'], row['G']), axis=1)

print(win\_percs, '\n')

## Writing Efficient Code with pandas

### Selecting columns and rows efficiently

* Measuring time:



* Always use build-in!

#### Locate rows: .iloc[] and .loc[]

* iloc faster than loc with *rows*.
* direct choosing works faster than iloc with *columns*.

### Replacing values in a DataFrame

#### Replace scalar values using .replace()

* Replacing values in rows:

poker\_hands['Class'].replace(1, -2, inplace=True)

#### Replace values using lists

* Replacing multiple values at the same time:
  + Multiple with one:

names['Ethnicity'].replace(['BLACK NON HISP', 'BLACK NON HISPANIC', 'WHITE NON HISP' , 'WHITE NON HISPANIC'], 'NON HISPANIC', inplace=True)

* + Multiple with multiple:

start\_time = time.time()

names['Ethnicity'].replace(['ASIAN AND PACI', 'BLACK NON HISP', 'WHITE NON HISP'],

          ['ASIAN AND PACIFIC ISLANDER', 'BLACK NON HISPANIC', 'WHITE NON HISPANIC'],

                            inplace=True)

print("Time using .replace(): {} sec".format(time.time() - start\_time))

#### Replace values using dictionaries

* Replacing with dictionaries:

poker\_hands.replace({'Explanation': {'Royal flush': 'Flush',

                                    'Straight flush': 'Flush'}}, inplace=True)

print(poker\_hands['Explanation'].head())

### Efficient iterating

#### Vectorization over pandas series

* Vectorization with **Series**:

mean\_r = poker\_hands.iloc[:, 1::2].mean(axis=1)

* Vactorization with **ndarray**:

start\_time = time.time()

poker\_var = poker\_hands[['R1', 'R2', 'R3', 'R4', 'R5']].values.var(axis=1, ddof=1)

print("Time using NumPy vectorization: {} sec".format(time.time() - start\_time))

print(poker\_var[0:5])

### Data manipulation using .groupby()

#### Data transformation using .groupby().transform

* Transform – conducts an aggregate to each group (works as partition):

min\_max\_tr = lambda x: (x - x.min()) / (x.max() - x.min())

restaurant\_grouped = restaurant\_data.groupby('time')

restaurant\_min\_max\_group = restaurant\_grouped.transform(min\_max\_tr)

print(restaurant\_min\_max\_group.head())

* More sofisticated transformation:

zscore = lambda x: (x - x.mean()) / x.std()

poker\_grouped = poker\_hands.groupby('Class')

poker\_trans = poker\_grouped.transform(zscore)

poker\_regrouped = poker\_trans.groupby(poker\_hands['Class'])

print(np.round(poker\_regrouped.mean(), 3))

print(poker\_regrouped.std())

#### Missing value imputation using transform()

* Imputation:

missing\_trans = lambda x: x.fillna(x.median())

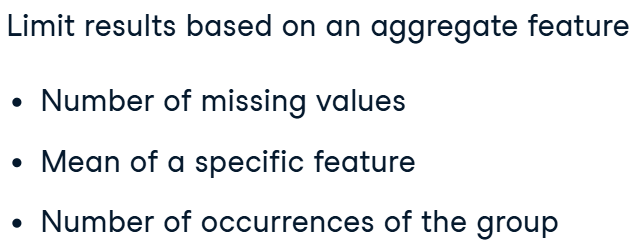
restaurant\_grouped = restaurant\_data.groupby('time')

restaurant\_impute = restaurant\_grouped.transform(missing\_trans)

print(restaurant\_impute.head())

#### Data filtration using the filter() function

* Purpose of filter():



* Code for filtering groupes based on different conditions:

total\_bill\_40 = restaurant\_data.groupby('day').filter(lambda x: x['total\_bill'].count() > 40)

total\_bill\_20 = total\_bill\_40.groupby('day').filter(lambda x: x['total\_bill'].mean() > 20)

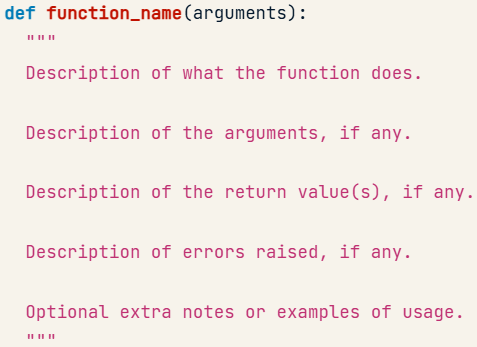
print('Days of the week that have a mean total\_bill greater than $20:', total\_bill\_20.day.unique())

## Writing Functions in Python

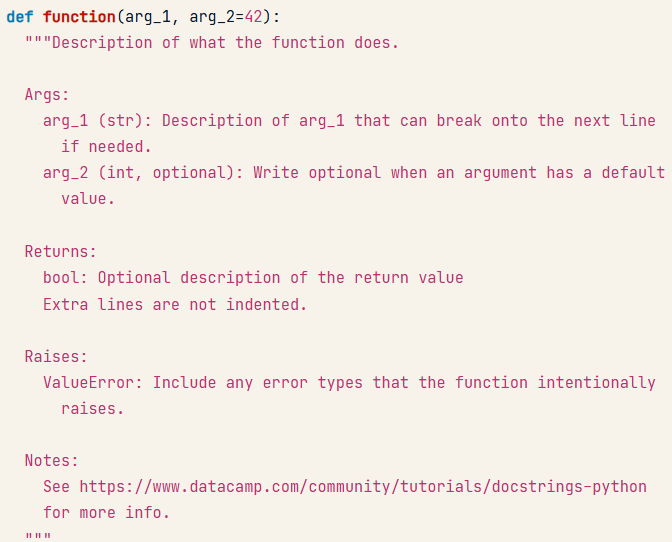
### Best Practices

#### Docstrings

* Anatomy of docstrings:



* Google style description:



* Code with docstrings:

def count\_letter(content, letter):

  """Count the number of times `letter` appears in `content`.

  Args:

    content (str): The string to search.

    letter (str): The letter to search for.

  Returns:

    int

  Raises:

    ValueError: raise an error when the arguments are not correct

  """

  if (not isinstance(letter, str)) or len(letter) != 1:

    raise ValueError('`letter` must be a single character string.')

  return len([char for char in content if char == letter])

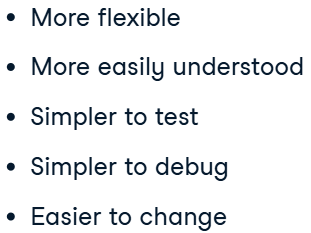
* Retrieving docstrings:

import inspect

inspect.getsource(function)

#### DRY and "Do One Thing"

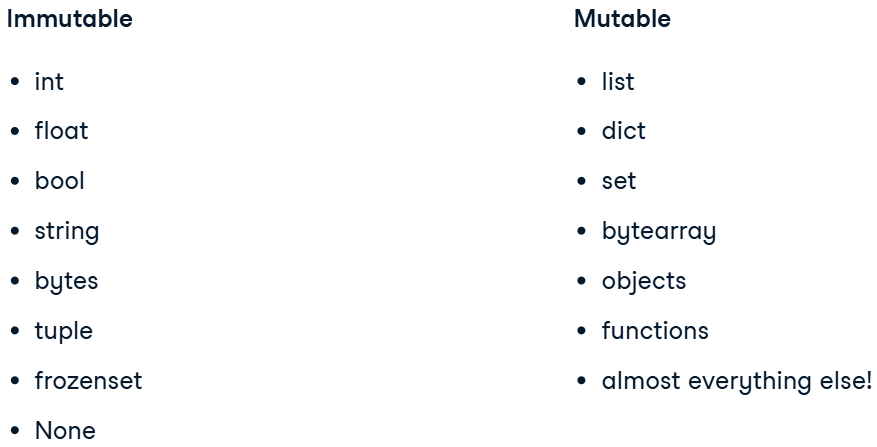
* Why we need to **create only one function for each task**:



* Anatomy of variable assigning:



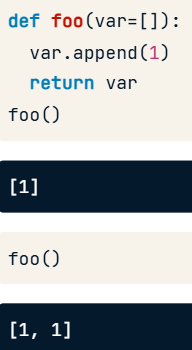
* Mutable or immutable:



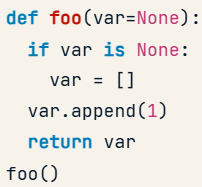
* + Notation:

The only way to tell if something is mutable is to see if there is a function or method that will change the object without assigning it to a new variable.

* + The problem of mutable variables (as a default variable):



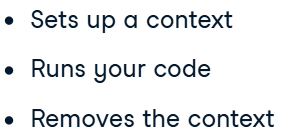
* + How to eliminate this (use as a defualt immutable variable):



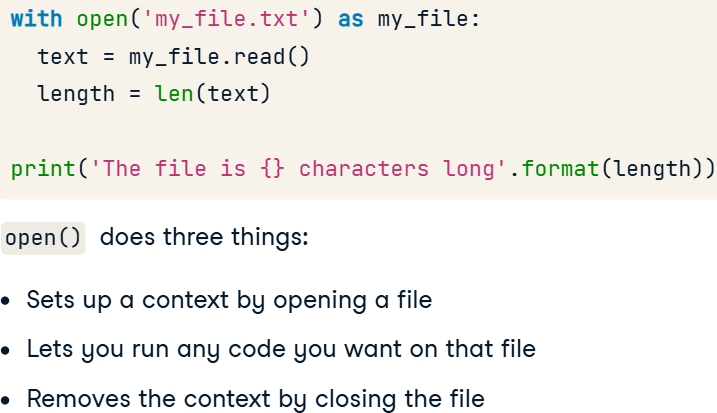
### Context Managers

#### Using context managers

* A context manager:

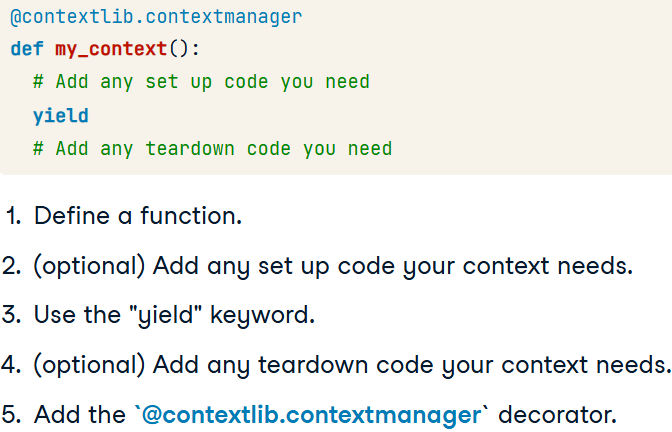


* + The process:



#### Writing context managers

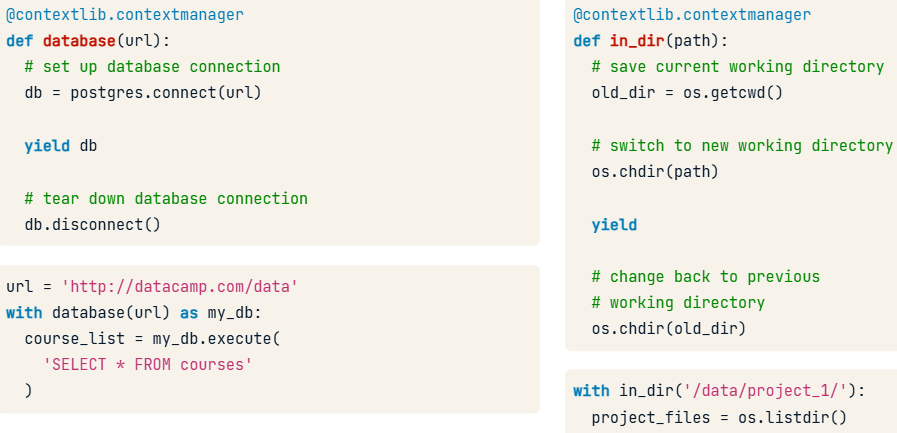
* The process:



* + Example:



* Code:



* Code and explonataion:

@contextlib.contextmanager

def open\_read\_only(filename):

  read\_only\_file = open(filename, 'r')

  yield read\_only\_file

  read\_only\_file.close()

with open\_read\_only('my\_file.txt') as my\_file:

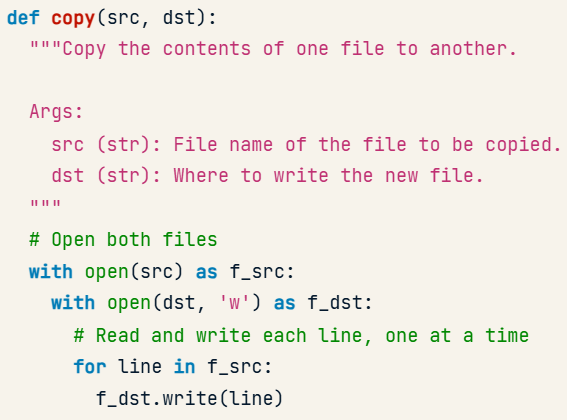
  print(my\_file.read())

* + Explonation:

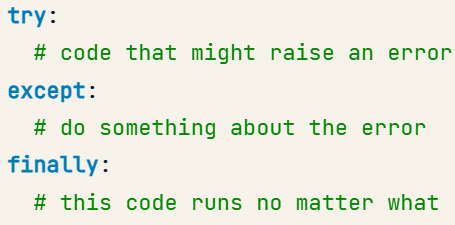
This function is an example of a context manager that \_does\_ return a value, so we write yield read\_only\_file instead of just yield. Then the read\_only\_file object gets assigned to my\_file in the with statement so that whoever is using your context can call its .read() method in the context block.

#### Advanced topics

* Nested contexts:



* Error handling:



* Example of code:

def in\_dir(directory):

  """Change current working directory to `directory`,

  allow the user to run some code, and change back.

  Args:

    directory (str): The path to a directory to work in.

  """

  current\_dir = os.getcwd()

  os.chdir(directory)

  try:

    yield

  finally:

    os.chdir(current\_dir)

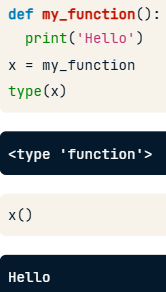
### Decorators

#### Functions are objects

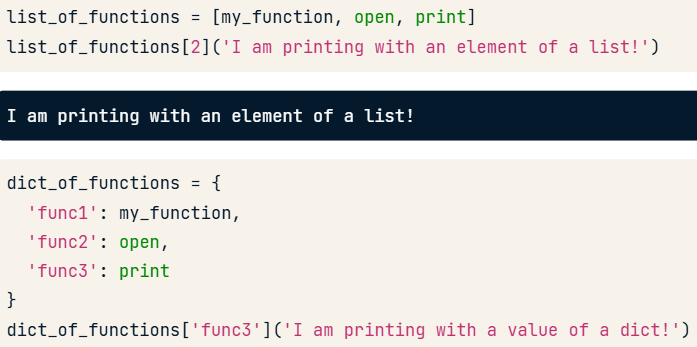
* Function – is an object.
* Referencing the function:



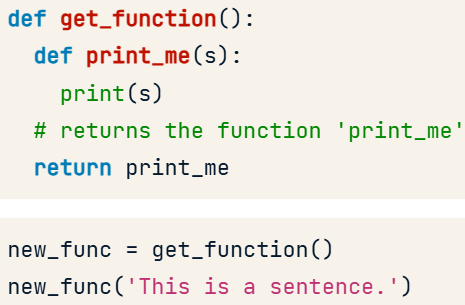
* Function as a variable:



* Lists and dictionaries as a functions:

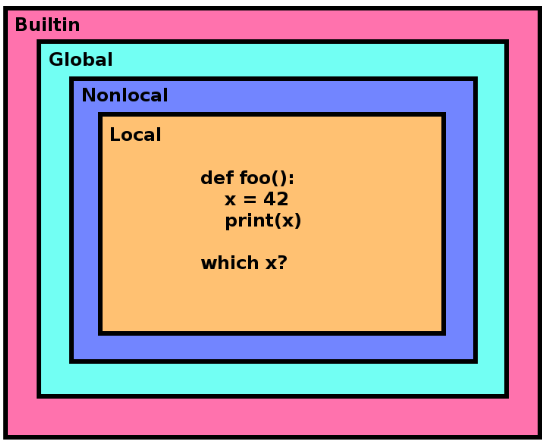


* Functions inside the functions:
  + new\_func is a “print\_me”, so it acts as a “print\_me”:



#### Scope

* All elements of the scope:
  + Порядок поиска: **LEGB** (Local → NonLocal → Global → Built-in).
    - **Built-in**: встроенные в Python функции и объекты, доступные всегда.
    - **Global**: переменные, объявленные на уровне модуля.
    - **Nonlocal**: переменные во вложенных функциях, но не глобальные.
    - **Local**: переменные, объявленные внутри функции или блока.



* Global:

def wait\_until\_done():

  def check\_is\_done():

    global done

    if random.random() < 0.1:

      done = True

  while not done:

    check\_is\_done()

done = False

wait\_until\_done()

print('Work done? {}'.format(done))

* Nonlocal:

def read\_files():

  file\_contents = None

  def save\_contents(filename):

    nonlocal file\_contents

    if file\_contents is None:

      file\_contents = []

    with open(filename) as fin:

      file\_contents.append(fin.read())

  for filename in ['1984.txt', 'MobyDick.txt', 'CatsEye.txt']:

    save\_contents(filename)

  return file\_contents

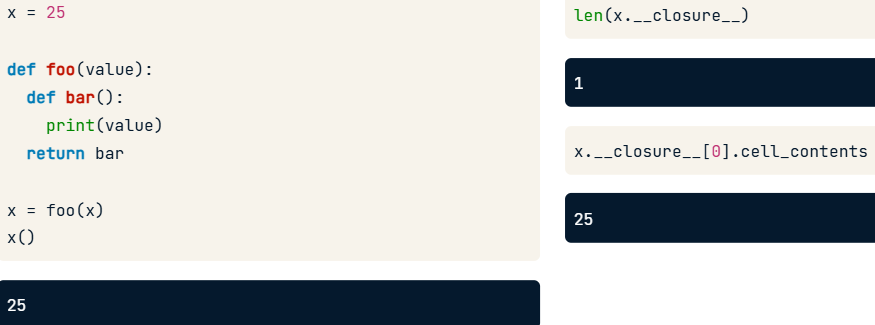
print('\n'.join(read\_files()))

#### Closures

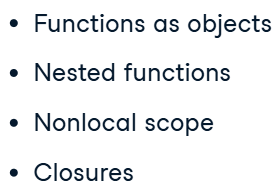
* Closure:

A closure in Python is a tuple of variables that are no longer in scope, but that a function needs in order to run.

* + Example:



* + - Even overwriting this var does’t loose the value, as it in closure.
* What does decorator use:



* Code for \_\_closure\_\_ checking:

def return\_a\_func(arg1, arg2):

  def new\_func():

    print('arg1 was {}'.format(arg1))

    print('arg2 was {}'.format(arg2))

  return new\_func

my\_func = return\_a\_func(2, 17)

print(my\_func.\_\_closure\_\_ is not None)

print(len(my\_func.\_\_closure\_\_) == 2)

closure\_values = [

  my\_func.\_\_closure\_\_[i].cell\_contents for i in range(0, 2)

]

print(closure\_values == [2, 17])

* + Additional explonation:

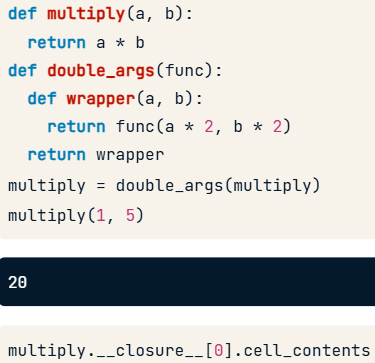
Values get added to a function's closure in the order they are defined in the enclosing function (in this case, arg1 and then arg2), but only if they are used in the nested function. That is, if return\_a\_func() took a third argument (e.g., arg3) that wasn't used by new\_func(), then it would not be captured in new\_func()'s closure.

#### Decorators

* Decorator:

Decorators are just functions that take a function as an argument and return a modified version of that function.

* Decorator and closures:



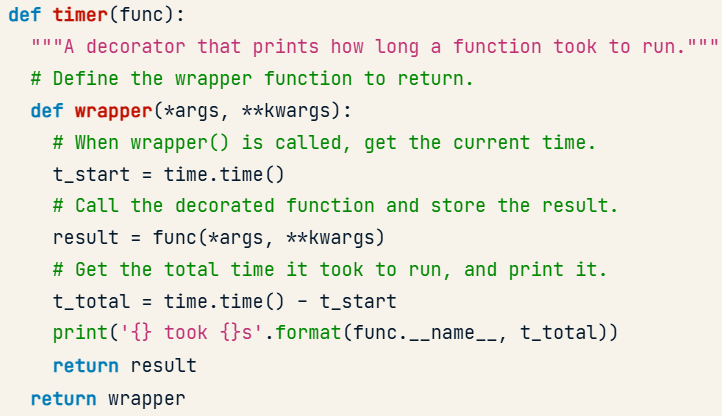
* Decorator syntax:



### More on Decorators

#### Real-world examples

* Code for decorator:



* How many times the function was called:

def counter(func):

  def wrapper(\*args, \*\*kwargs):

    wrapper.count += 1

    result = func(\*args, \*\*kwargs)

    return result

  wrapper.count = 0

  return wrapper

@counter

def foo():

  print('calling foo()')

foo()

foo()

print('foo() was called {} times.'.format(foo.count))

#### Decorators and metadata

* Saving metadata of our initial function, not the “wrapper” function:

from functools import wraps

def add\_hello(func):

  @wraps(func)

  def wrapper(\*args, \*\*kwargs):

    """Print 'hello' and then call the decorated function."""

    print('Hello')

    return func(\*args, \*\*kwargs)

  return wrapper

@add\_hello

def print\_sum(a, b):

  """Adds two numbers and prints the sum"""

  print(a + b)

print\_sum(10, 20)

print\_sum\_docstring = print\_sum.\_\_doc\_\_

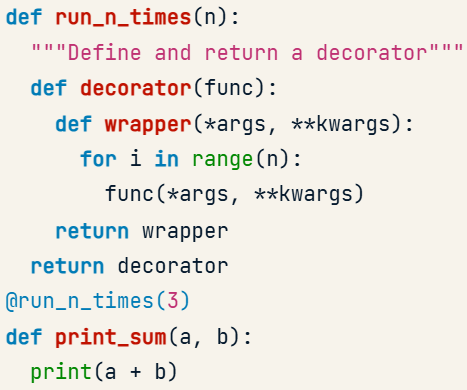
print(print\_sum\_docstring)

* Calling the original function:

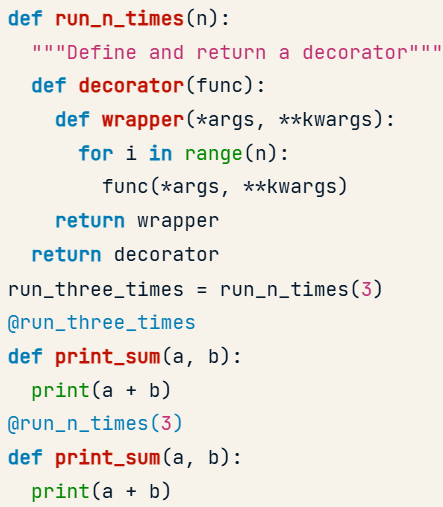
duplicate.\_\_wrapped\_\_(list(range(50)))

#### Decorators that take arguments

* Creating the decorator which takes arguments:



* The process:



* Code:

from functools import wraps

def returns(dtype):

  def decorator(func):

    @wraps(func)

    def wrapper(\*args, \*\*kwargs):

      result = func(\*args, \*\*kwargs)

      assert type(result) == dtype

      return result

    return wrapper

  return decorator

@returns(dict)

def foo(value):

  return value

try:

  print(foo([1,2,3]))

except AssertionError:

  print('foo() did not return a dict!')

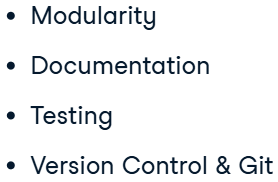
* + assert is a keyword that you can use to test whether something is true. If you type assert condition and condition is True, this function doesn't do anything. If condition is False, this function raises an AssertionError.

## Software Engineering Principles in Python

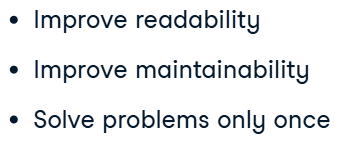
### Software Engineering & Data Science

#### Python, data science, & software engineering

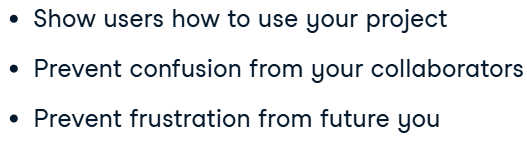
* Concepts:



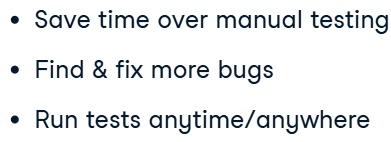
* Modularity:



* + **Ways of writing it: packages**, **classes**, and **methods.**
* Documentation:



* Automated testing:



#### Conventions and PEP 8

* Package for cheking congruence of our code (file) to ***PEP8***: pycodestyle.
* Code with files:

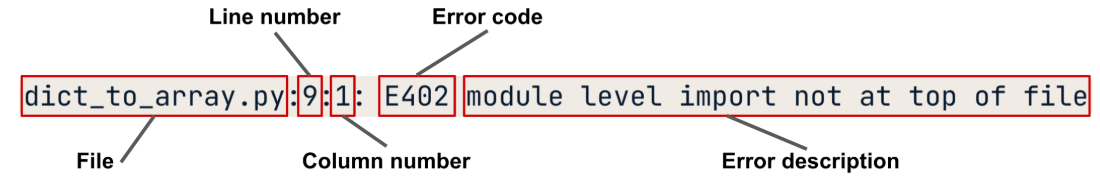
import pycodestyle

style\_checker = pycodestyle.StyleGuide()

result = style\_checker.check\_files(['nay\_pep8.py', 'yay\_pep8.py'])

print(result.messages)

* + The output:



* pylint file.py

### Writing a Python Module

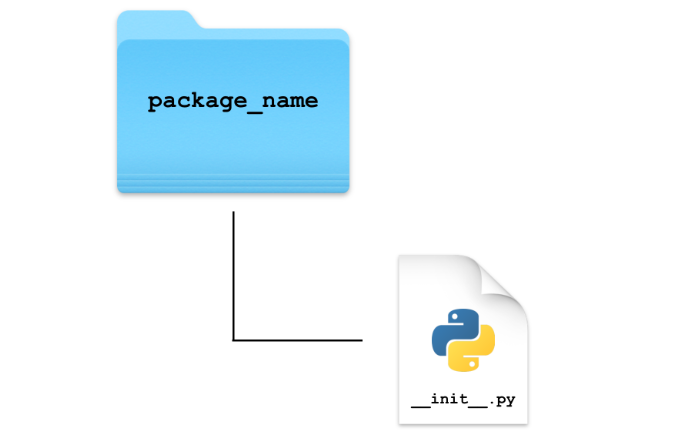
#### Writing your first package

* Package structure:

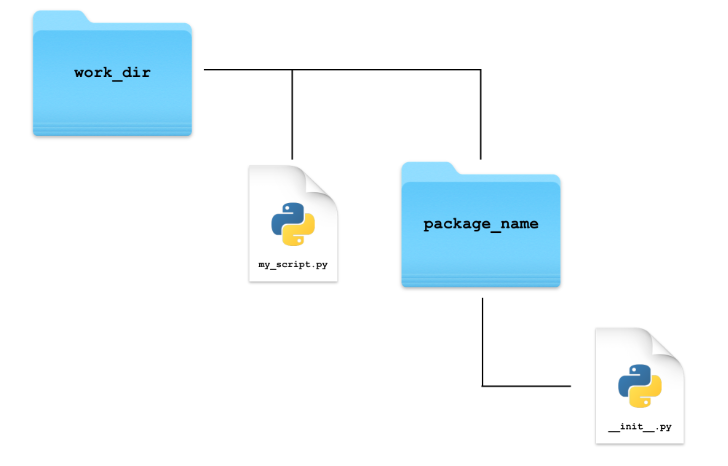
A minimal python package consists of 2 elements: a directory and a python file.

Name of the directory is a name of the package.

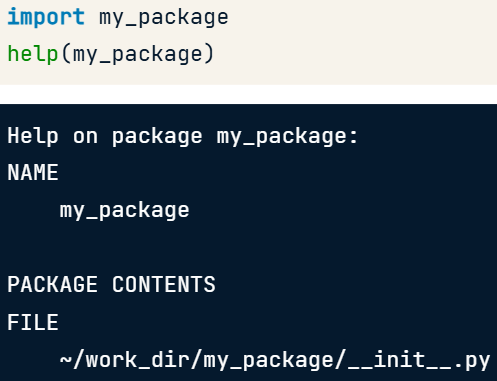
* + Structure:



* + With this structure we have created a package, that we can import as numpy or something else.
  + The *\_\_init\_\_.py* file lets Python know that a directory is a package.
* Importing a local package:

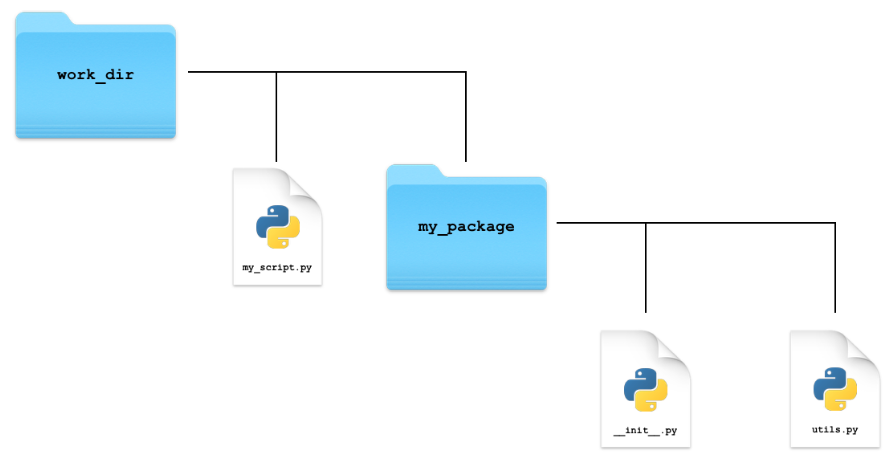


* Improting local package:

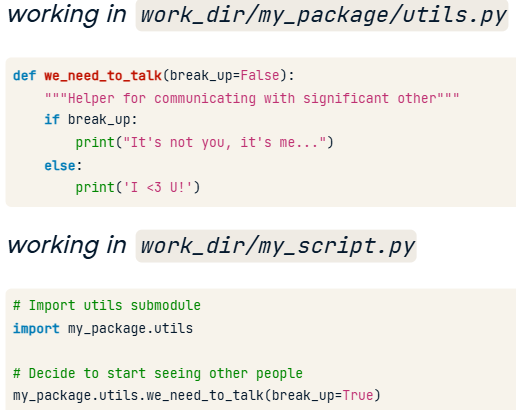


#### Adding functionality to packages

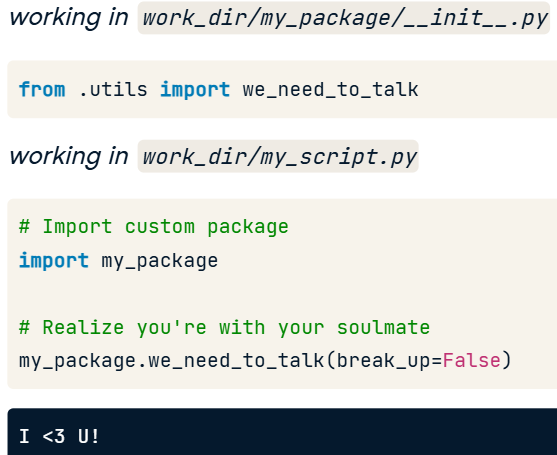
* Modification of the structure:



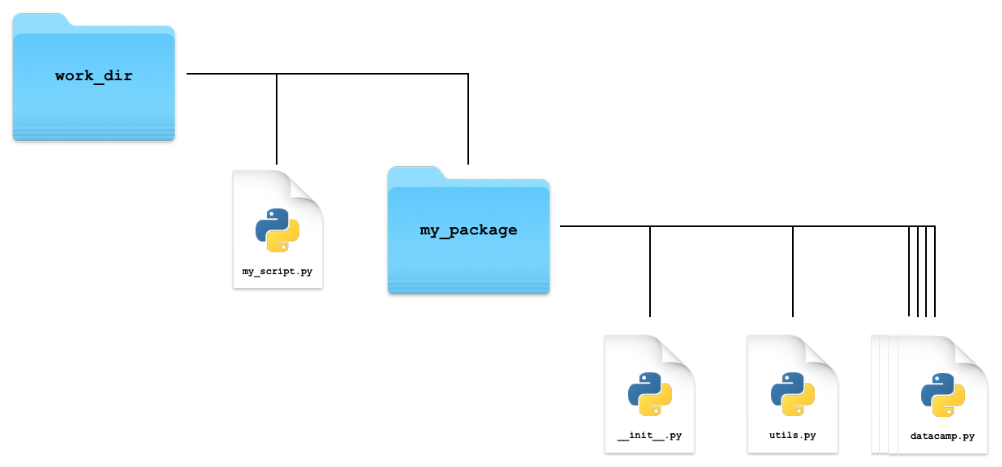
* + Code:



* Importing functionality in *\_\_init\_\_.py*:



* Extending functionality:



* + you should import your package's key functionality in your \_\_*init\_\_* file to make it directly and easily accessible.
* Extending package structure:

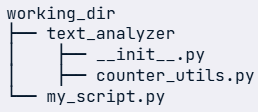


* Example of code working with added functions in *\_\_init\_\_.py*:

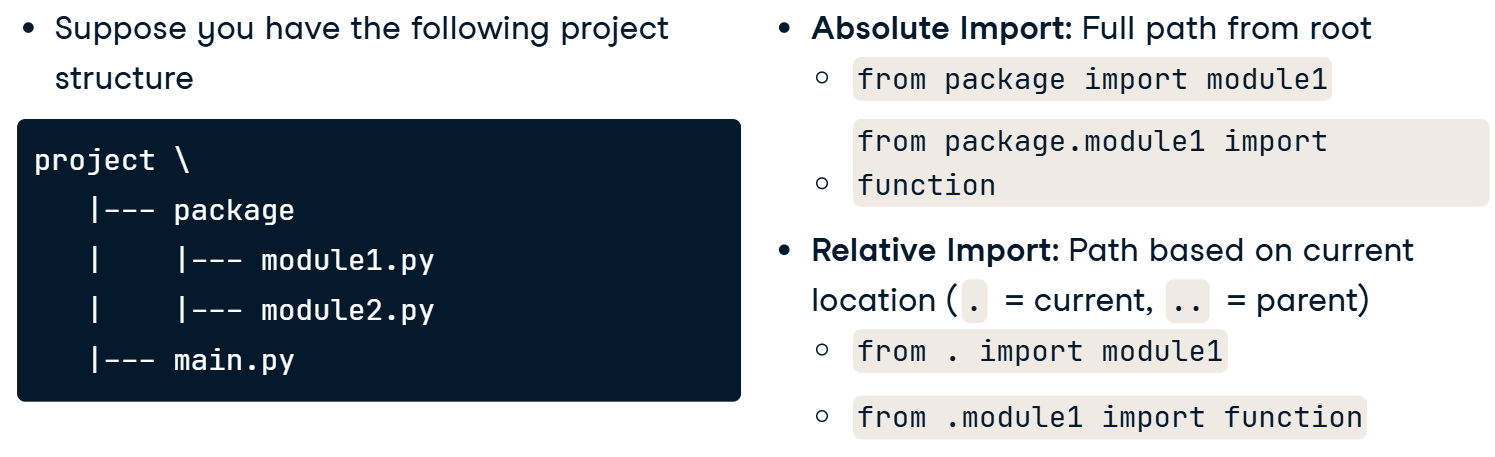
import text\_analyzer

word\_count\_totals = text\_analyzer.sum\_counters(word\_counts)

text\_analyzer.plot\_counter(word\_count\_totals)



* Relative and absolute importing:

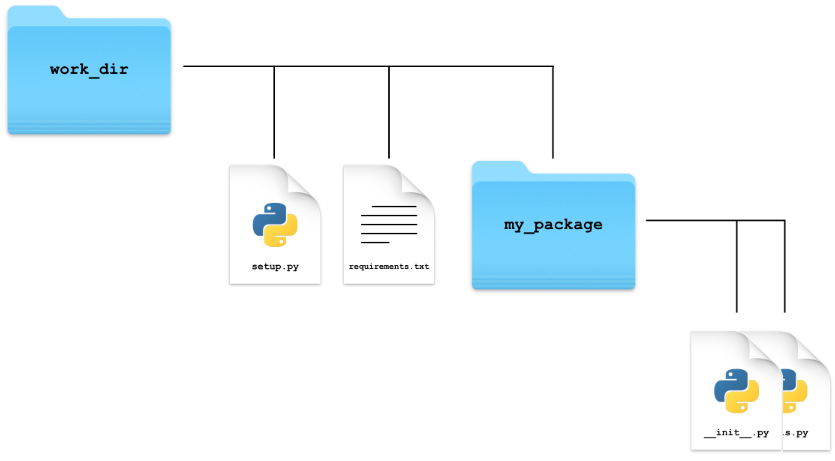


#### Making your package portable

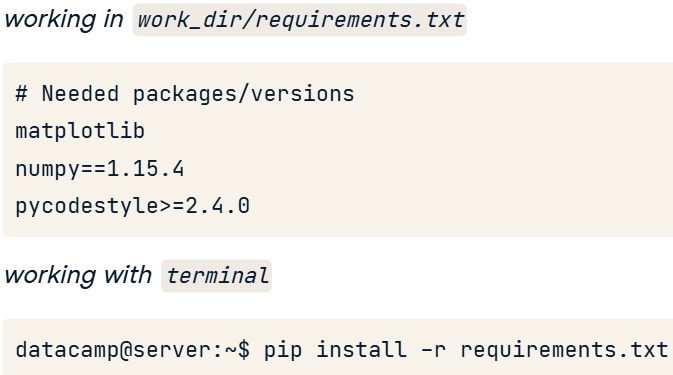
* Steps to portability (creating those two files):



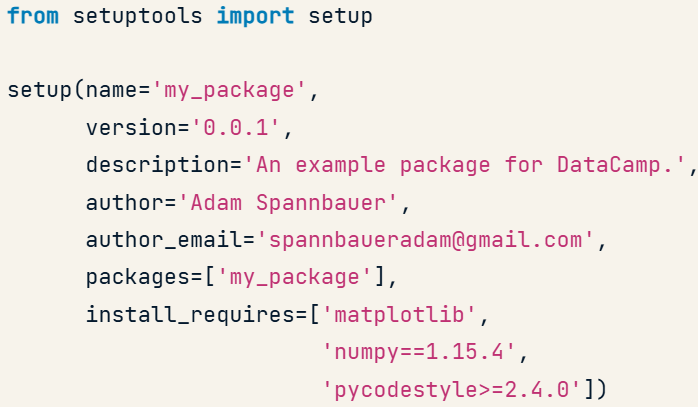
* Portable package structure:



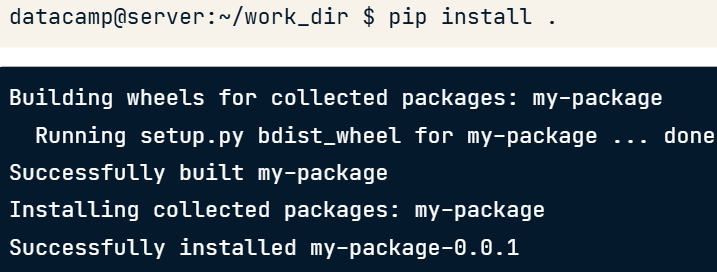
* Contents of *requirements.txt* (environment):
  + A requirements file shows how to recreate the environment needed to properly use your package.
  + This includes a list of python packages and optionally the version requirements for each package.
    - Process and setting the env up:



* Contents of *setup.py*:



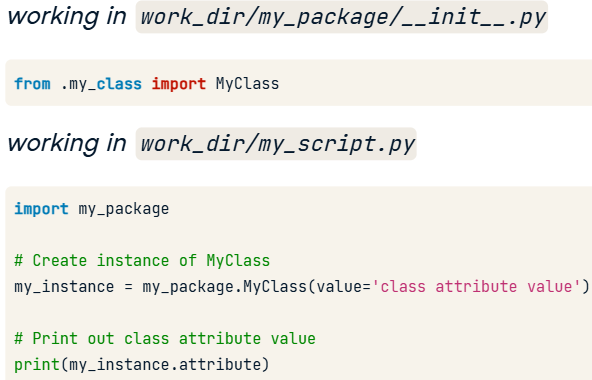
* + Packages argument has to include submodules, if they exist.
* Installing own package:



### Utilizing Classes

#### Adding classes to a package

* Using a *class* in a package:



#### Adding functionality to classes

* Creating non-public functions (adding “\_” before the name):

class Document:

  def \_\_init\_\_(self, text):

    self.text = text

    self.tokens = self.\_tokenize()

    self.word\_counts = self.\_count\_words()

  def \_tokenize(self):

    return tokenize(self.text)

  def \_count\_words(self):

    return Counter(self.tokens)

* + By defining methods as non-public you're signifying to the user that the method is only to be used inside the package.

#### Classes and the DRY principle

* Inheritance:

class SocialMedia(Document):

    def \_\_init\_\_(self, text):

        Document.\_\_init\_\_(self, text)

        self.hashtag\_counts = self.\_count\_hashtags()

        self.mention\_counts = self.\_count\_mentions()

    def \_count\_hashtags(self):

        return filter\_word\_counts(self.word\_counts, first\_char='#')

    def \_count\_mentions(self):

        return filter\_word\_counts(self.word\_counts, first\_char='@')

* Using code (class was imported in a \_\_init\_\_.py):

import text\_analyzer as ta

my\_doc = ta.Document(datacamp\_tweets)

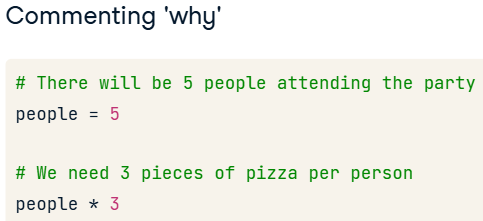
help(my\_doc.plot\_counts)

my\_doc.plot\_counts()

### Maintainability

#### Documentation

* *Effective comments* (answer the answer “why” your line of code is doing something):



* Using function with better commenting:

import re

def extract\_0(text):

    # match and extract dollar amounts from the text

    return re.findall(r'\$\d+\.\d\d', text)

def extract\_1(text):

    # return all matches to regex pattern

    return re.findall(r'\$\d+\.\d\d', text)

# Print the results of the function with better commenting

print(extract\_0(text))

#### Readability counts

* Descriptive naming:



* Functions have to accomplish only one thing, because it easy to *read and test*.
* Proper naming:

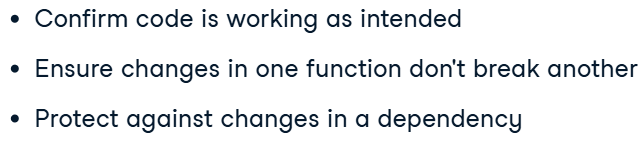
pupil\_diameter = [3.3, 6.8, 7.0, 5.4, 2.7]

mean\_diameter = mean(pupil\_diameter)

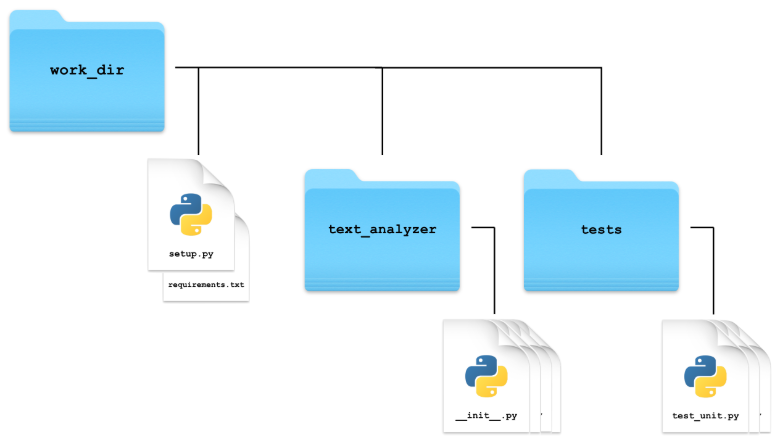
print(mean\_diameter)

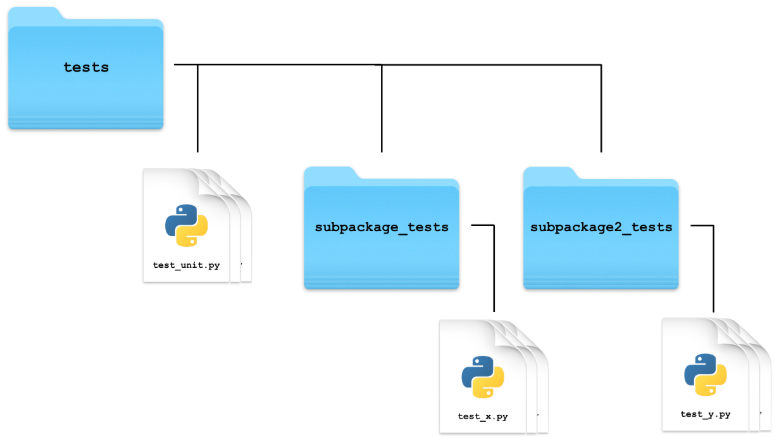
#### Unit testing

* Why testing:



* *PyTest* structure:





* + Pytest searches for tests by first looking for files that start or end with the word test, and then pytest runs all the functions in these files who's name follow the same pattern.
* Writing unit-testing:



* + Code:

from collections import Counter

from text\_analyzer import SocialMedia

test\_post = 'learning #python & #rstats is awesome! thanks @datacamp!'

sm\_post = SocialMedia(test\_post)

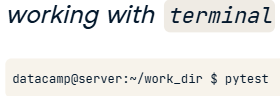
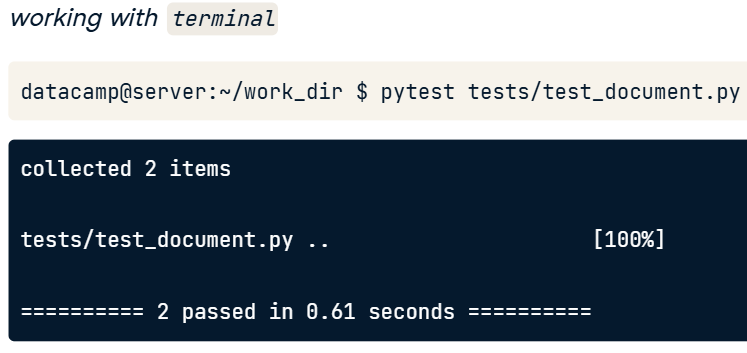
# Test hashtag counts are created properly

def test\_social\_media\_hashtags():

    expected\_hashtag\_counts = Counter({'#python': 1, '#rstats': 1})

    assert sm\_post.hashtag\_counts == expected\_hashtag\_counts

* Running pytest:



* Using *doctest* + documentation:

def sum\_counters(counters):

    """Aggregate collections.Counter objects by summing counts

    :param counters: list/tuple of counters to sum

    :return: aggregated counters with counts summed

    >>> d1 = text\_analyzer.Document('1 2 fizz 4 buzz fizz 7 8')

    >>> d2 = text\_analyzer.Document('fizz buzz 11 fizz 13 14')

    >>> sum\_counters([d1.word\_counts, d2.word\_counts])

    Counter({'fizz': 4, 'buzz': 2})

    """

    return sum(counters, Counter())

import doctest

doctest.testmod()

#### Documentation & testing in practice

* Documenting classes:

from text\_analyzer import Document

class SocialMedia(Document):

    """Analyze text data from social media

    :param text: social media text to analyze

    :ivar hashtag\_counts: Counter object containing counts of hashtags used in text

    :ivar mention\_counts: Counter object containing counts of @mentions used in text

    """

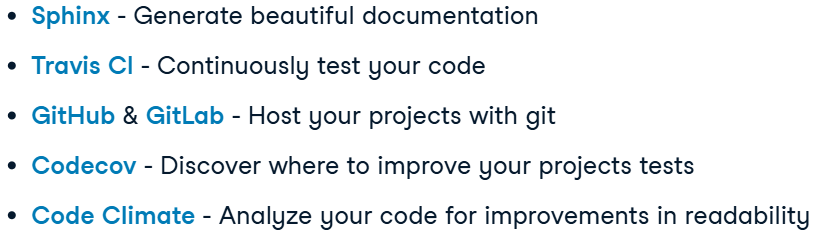
    def \_\_init\_\_(self, text):

        Document.\_\_init\_\_(self, text)

        self.hashtag\_counts = self.\_count\_hashtags()

        self.mention\_counts = self.\_count\_mentions()

* Links and additional tools for ejgineering:



## Introduction to Object-Oriented Programming in Python

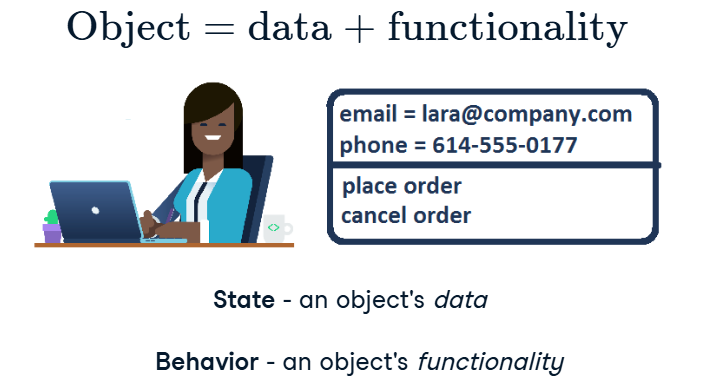
### OOP Fundamentals

#### What is OOP?

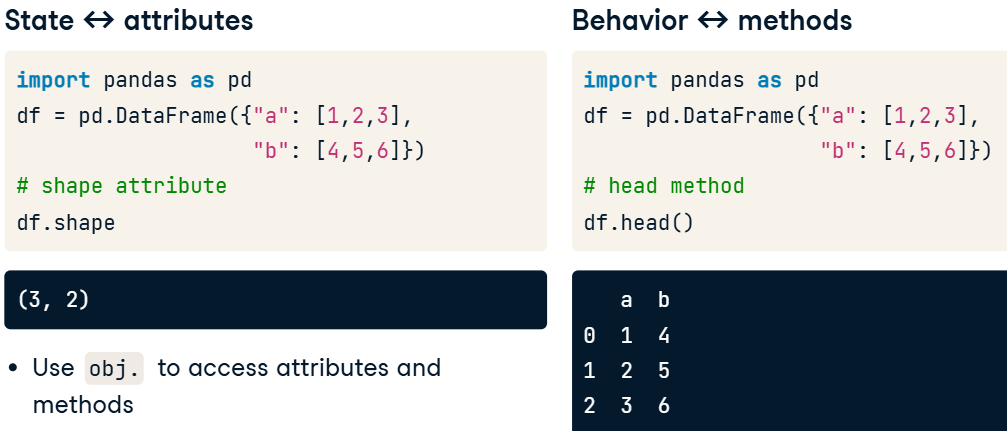
* Quick refresh:

https://campus.datacamp.com/courses/intermediate-object-oriented-programming-in-python/overloading-and-multiple-inheritance?ex=1

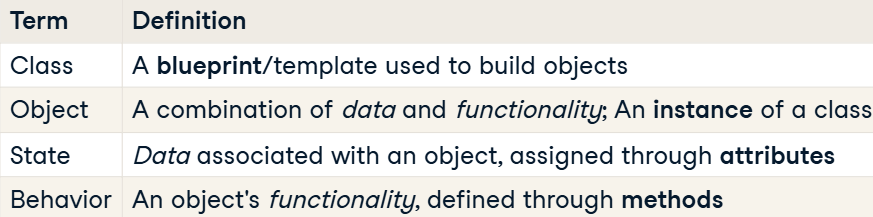
* Object:
  + Classes are like blueprints for objects. They describe the possible states and behaviors that every object from that class can have.



* Attributes and methods:
  + State – attribute (what does it have); behavior – method (what does it can).



* Cheat Sheet:



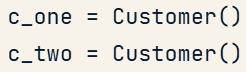
Classes and objects both have attributes and methods, but the difference is that a *class* is an abstract template, while an *object* is a concrete representation of a class.

* Displaying attributes and methods:
  + You can use dir() to list all attributes and methods for an object or class.

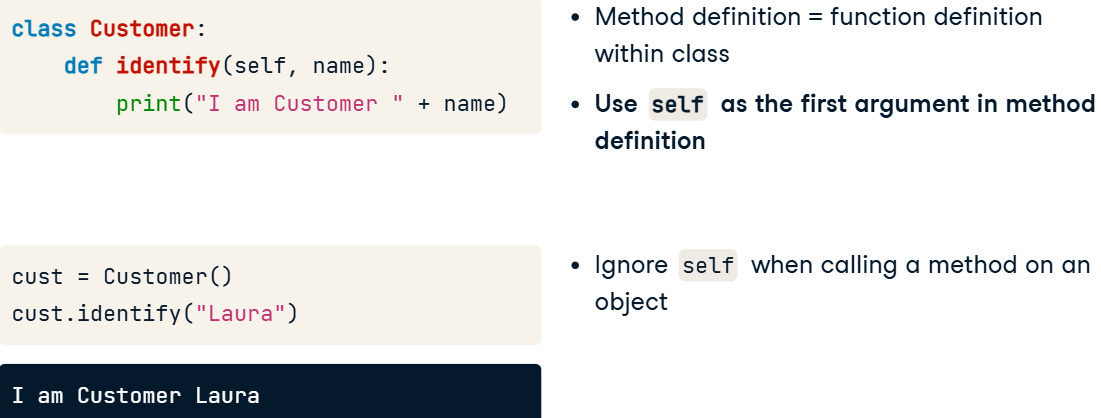
print(dir(float))

#### Class anatomy: attributes and methods

* Objects of the class:



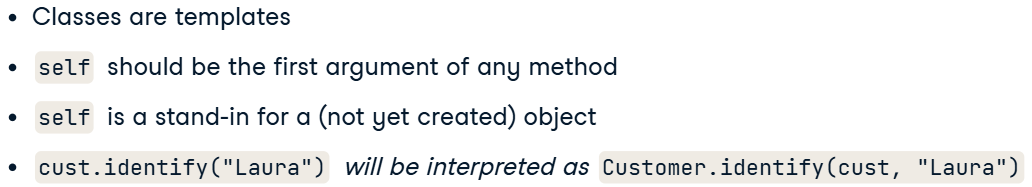
* Adding method to the Class:



* + Explonation:

You can use methods to print or return values, or to make plots, *as long as the behavior is appropriate for objects of that class*, e.g., an Employee probably wouldn't have a pivot\_table() method.

* What is **self**:



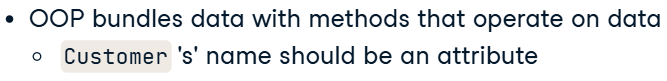
* + How (self) is working:

*Класс* — это чертёж.  
 *Объект* — это то, что сделано по этому чертежу.

*self* — это способ для объекта сказать: "Привет, это я! Работай со мной."

Когда вызывается *метод* *объекта*, Python автоматически передаёт *объект* в *self*, чтобы *метод* знал, с каким именно *объектом* он работает.

* Attribute:



* + Example: customer name should be an attribute of a customer object, instead of a parameter passed to a method.
  + Adding an attribute:



* Code (creating and using methods, attributes and using existing attributes in the code):

class Employee:

  def set\_name(self, new\_name):

    self.name = new\_name

  def set\_salary(self, new\_salary):

    self.salary = new\_salary

  def give\_raise(self, amount):

    self.salary += amount

emp = Employee()

emp.set\_name('Korel Rossi')

emp.set\_salary(50000)

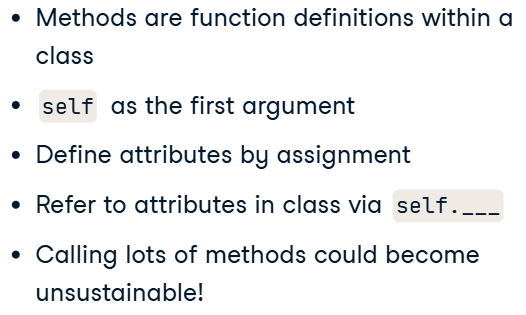
print(emp.salary)

emp.give\_raise(1500)

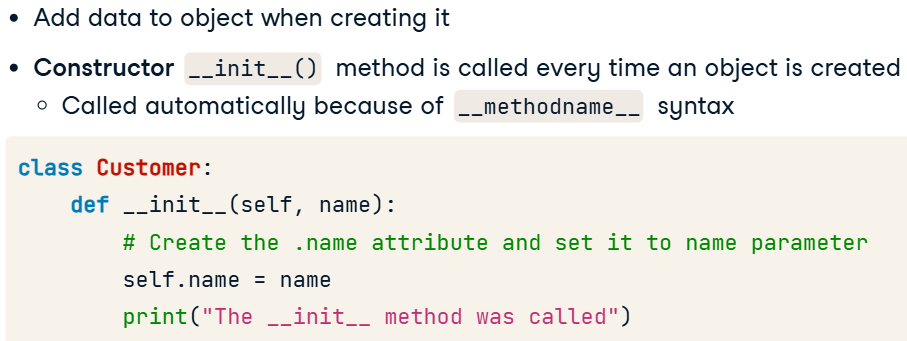
print(emp.salary)

#### Class anatomy: the \_\_init\_\_ constructor

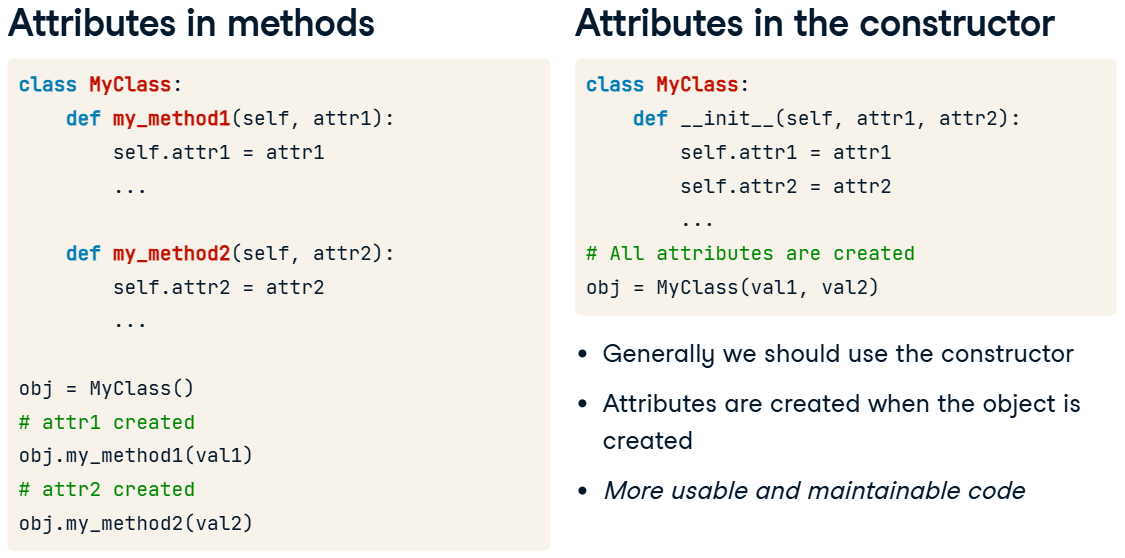
* Methods and attibutes and why we need *\_\_init\_\_*:



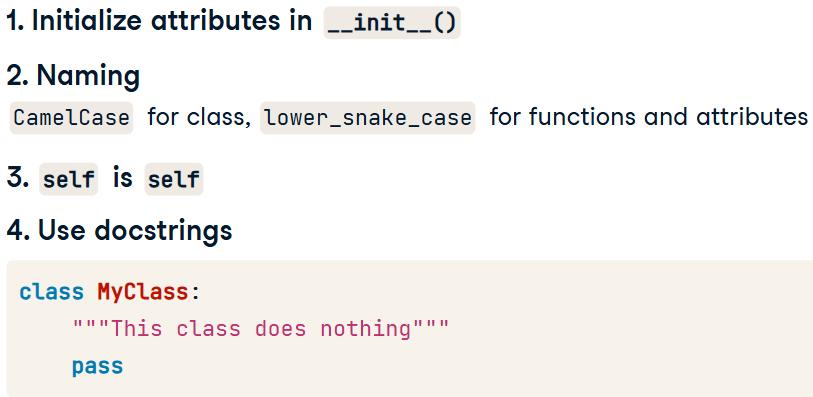
* Constructor:



* Attributes in methods / in constructor:



* Best practices:



* Code:

class Calculator:

  def \_\_init\_\_(self, num\_one, num\_two):

    self.num\_one = num\_one

    self.num\_two = num\_two

  def addition(self):

    return self.num\_one + self.num\_two

  def subtraction(self):

    return self.num\_one - self.num\_two

  def multiplication(self):

    return self.num\_one \* self.num\_two

### Inheritance and Polymorphism

#### Class vs. instance attributes

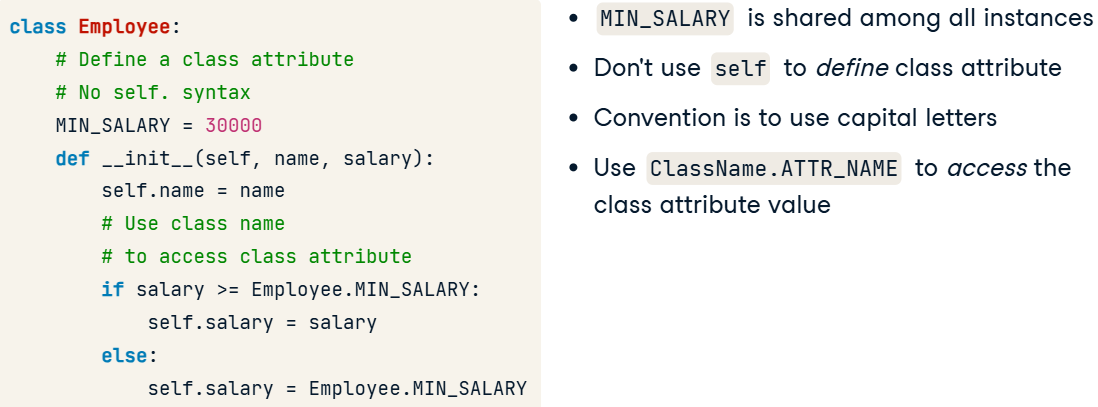
* Core principles of OOP:

1. **Encapsulation**: Combining data (attributes) and behavior (methods) into a single unit (a class), making it easier to manage and protect. Grouping related data and methods in a class and restricting direct access to internal details.  
    Think of it as a way to organize and control access to related data and actions.

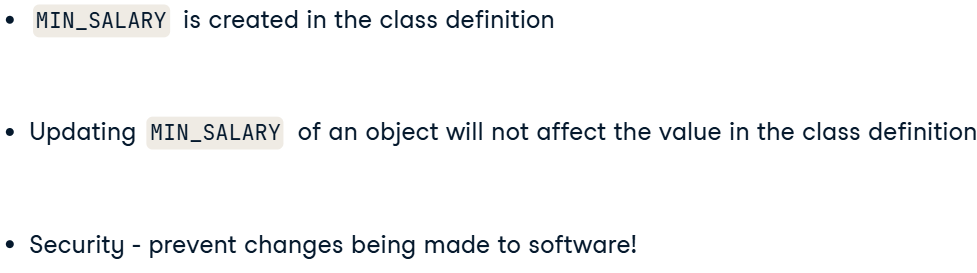
У тебя есть пульт. Ты нажимаешь кнопку "включить", и телевизор включается. Как он это делает? Неважно. Главное — тебе дали кнопку, а все сложное спрятали внутри.

1. **Inheritance**: Reusing and extending existing code by creating new classes based on existing ones.  
   It's like a child inheriting traits from a parent but also adding unique ones.
2. **Polymorphism**: Using the same *method name* or interface to work with different types of objects.  
   It lets you treat different objects from the same family in the same way.

* Instance-level attribute:
  + Can access the specific value of attribute of an object.
* Class-level attribute:
  + Data shares among all instances of a class (min salary in state);
  + Define *class attributes* in the body;
  + “Global variable” within the class.
* Code:



* Modifying class-level attributes;



* + Code:

p1 = Player(9)

p2 = Player(5)

print("MAX\_POSITION of p1 and p2 before assignment:")

print(p1.MAX\_POSITION)

print(p2.MAX\_POSITION)

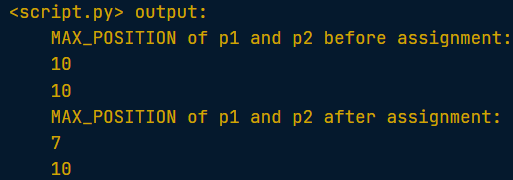
p1.MAX\_POSITION = 7

print("MAX\_POSITION of p1 and p2 after assignment:")

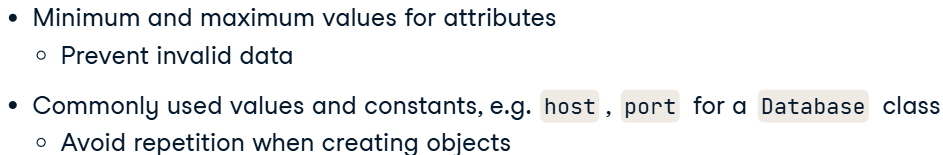
print(p1.MAX\_POSITION)

print(p2.MAX\_POSITION)

result:

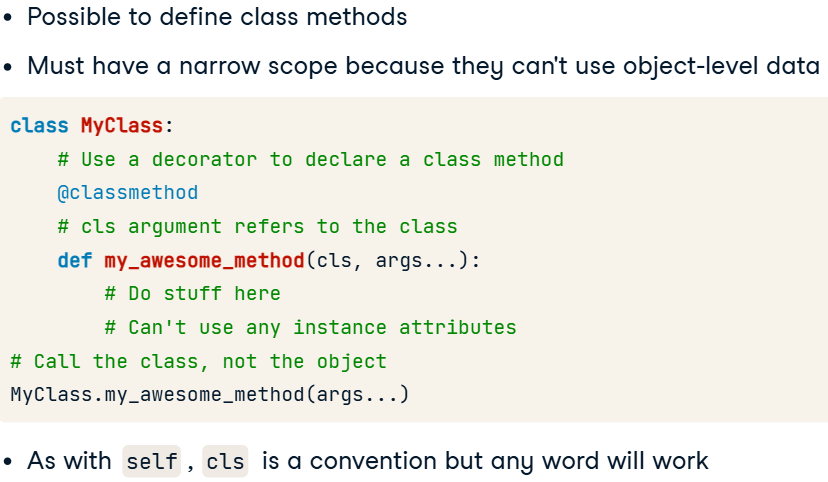


* + If you want to change the value of the class attribute at runtime, you need to do it by referring to the class name, not through an instance.
* Why use class-attributes?

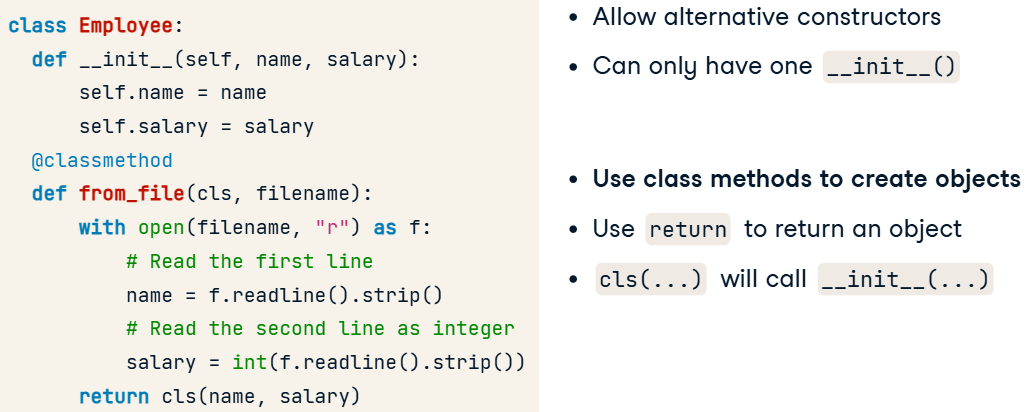


#### Class methods

* Creating class-method:



* + Class methods are a great way of allowing an alternative way to create an object from a class, such as by a file or by accepting different information and performing a task during the construction to return the required attributes.
* Alternative constuctors (why use class-methods):



* When to use class-methods:

1. **Альтернативные конструкторы**:  
   Например, для создания объектов из файлов или других форматов.
2. **Методы, не зависящие от объекта**:  
   Когда метод не требует работы с атрибутами конкретного объекта.
3. **Singleton (Одиночка)**:  
   Чтобы ограничить класс одним объектом, например:
   * Для подключения к базе данных.
   * Для хранения настроек.

* Code:

class BetterDate:

  def \_\_init\_\_(self, year, month, day):

    self.year, self.month, self.day = year, month, day

  @classmethod

  def from\_str(cls, datestr):

    parts = [int(i) for i in datestr.split("-")]

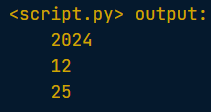
    year, month, day = parts

    return cls(year, month, day)

xmas = BetterDate.from\_str("2024-12-25")

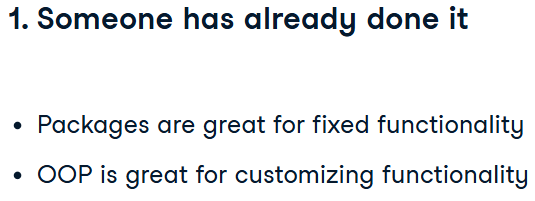
print(xmas.year)

result:



#### Class inheritance

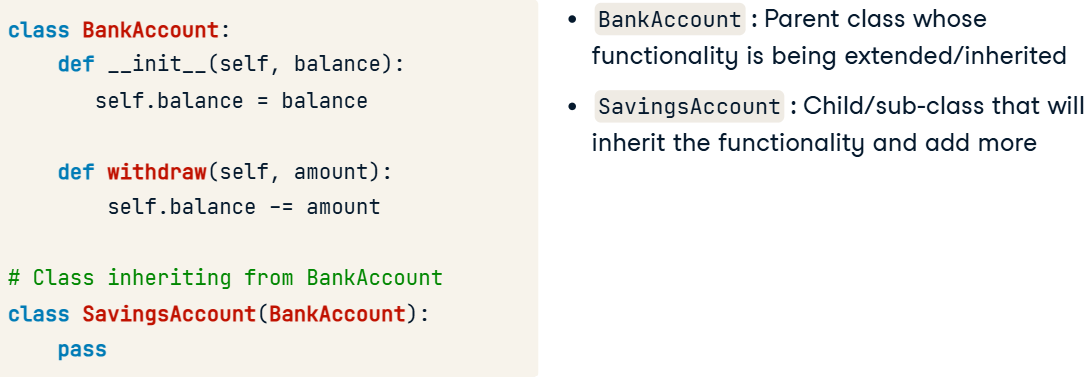
* Code reuse:



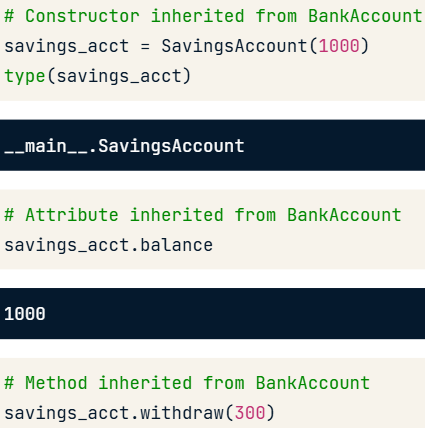
* Inheritance:



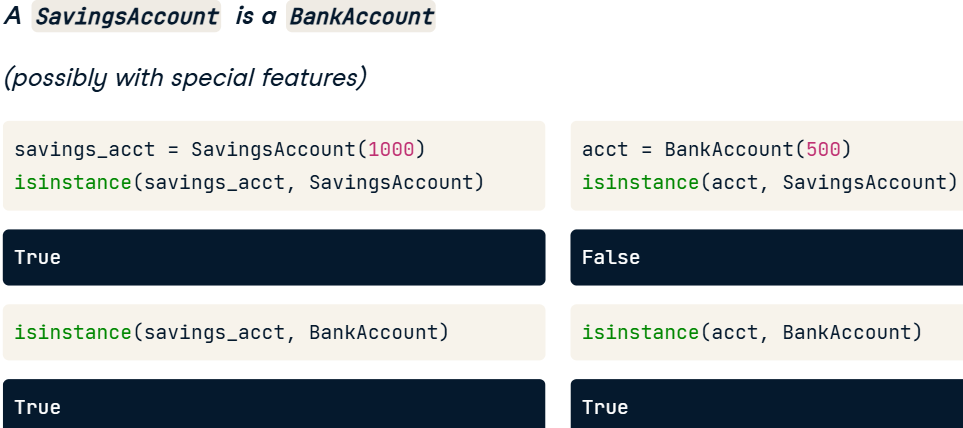
* Implementing class inheritance (creating subclasses):
  + The purpose of child classes, or sub-classes, is to customize and extend the functionality of the parent class.



* + We can access attributes and methods without even creating those in a new class:



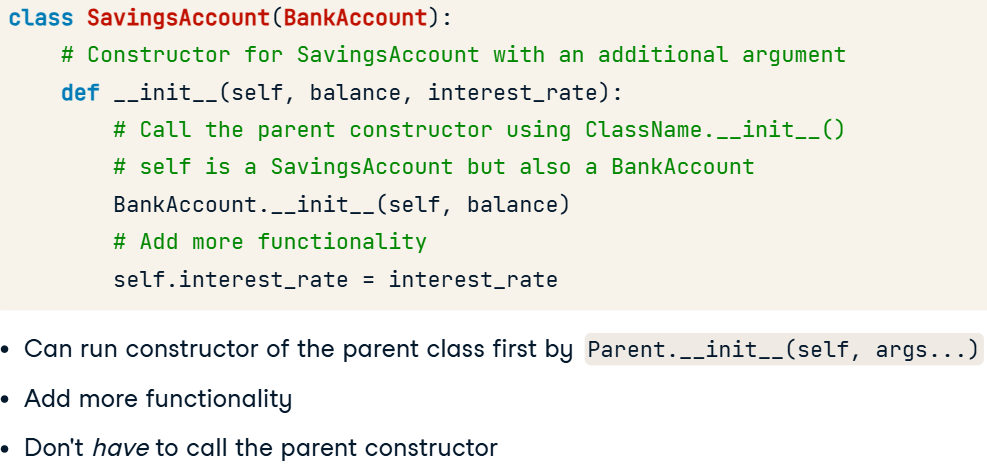
* + This is an example of both *polymorphism*, which defines a unified interface, as well as *inheritance*, because the parent and child classes can be used in the same way!
* Is-a relationship (objects of SA is-a objects of BA):



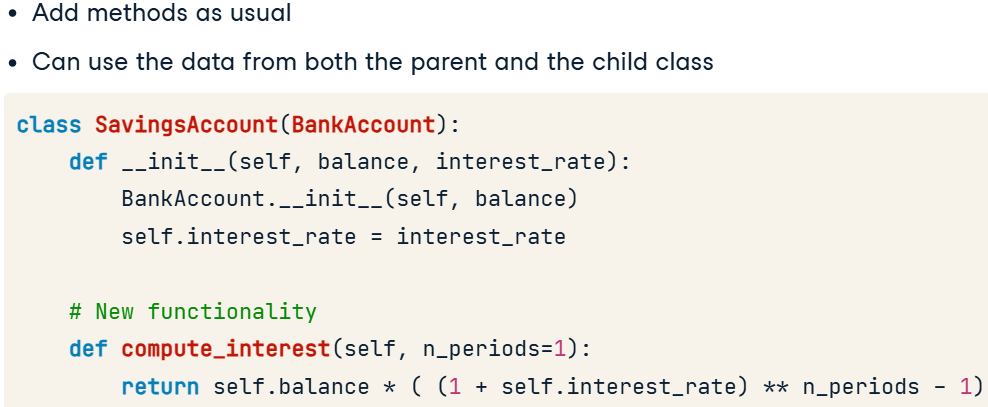
* + Additional explonation: The fact that the instances of a child class are also instances of the parent class allows you to create consistent interfaces. Any place that a Counter could go - for example, as an argument to a function, you will be able to use Indexer instead because it has the same methods and attributes as Counter.

#### Customizing functionality via inheritance

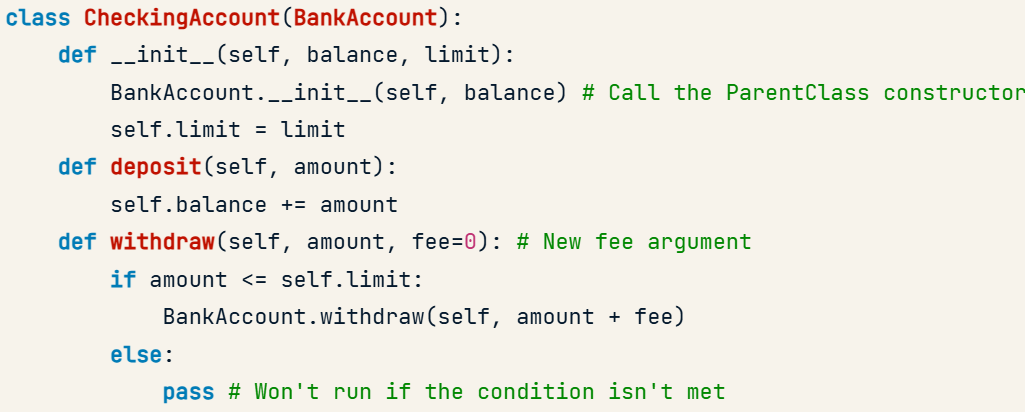
* Customizing constructors:



* Adding functionality:



* + SavingsAccount inherits the withdraw method from the parent BankAccount class, so calling withdraw on a SavingsAccount object will execute the same code as calling it on a BankAccount object.
* Second child class creation:



* + It's worth noting that our approach technically *violates polymorphism* because the parent and child class no longer have a *unified interface* - their methods have different arguments, so they do not work in the same way.
* Code:

class Employee:

  def \_\_init\_\_(self, name, salary=30000):

    self.name = name

    self.salary = salary

  def give\_raise(self, amount):

    self.salary += amount

class Manager(Employee):

  def \_\_init\_\_(self, name, salary=50000, project=None):

    Employee.\_\_init\_\_(self, name, salary)

    self.project = project

def display(self):

    print("Manager ", self.name)

  def give\_raise(self, amount, bonus=1.05):

    new\_amount = amount\*bonus

    Employee.give\_raise(self, new\_amount)

mngr = Manager("Ashta Dunbar", 78500)

mngr.give\_raise(2000, bonus=1.03)

print(mngr.salary)

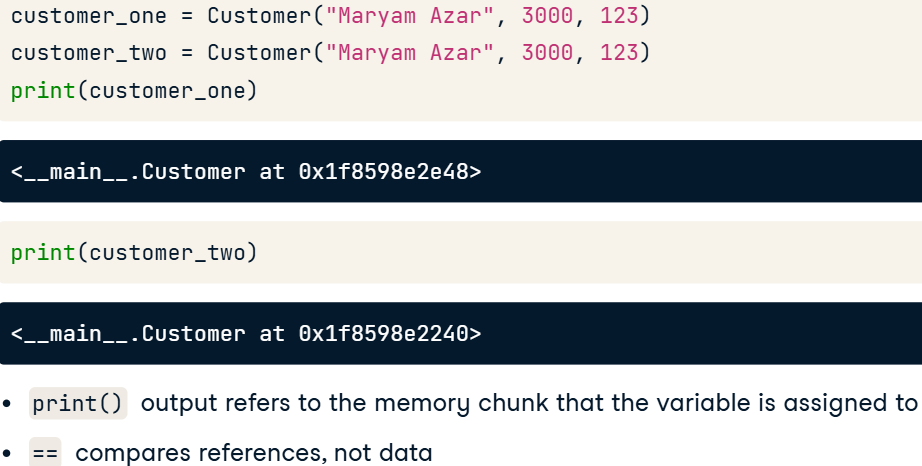
result:



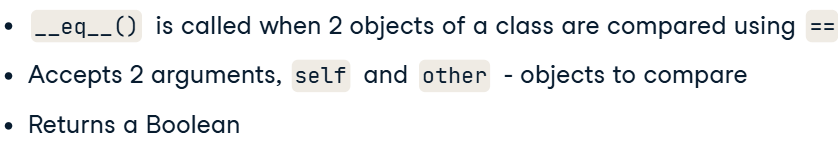
### Integrating with Standard Python

#### Operator overloading: comparing objects

* Variables are references:



* The \_\_eq\_\_ method:



* + Code:

class BankAccount:

  def \_\_init\_\_(self, number, balance=0):

    self.balance = balance

    self.number = number

  def withdraw(self, amount):

    self.balance += amount

  def \_\_eq\_\_(self, other):

    return (self.number == other.number)

acct1 = BankAccount(123, 1000)

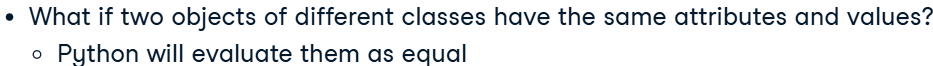
acct2 = BankAccount(123, 1000)

acct3 = BankAccount(456, 1000)

print(acct1 == acct2)

print(acct1 == acct3)

* Checking equality:



* + Code:

class BankAccount:

  def \_\_init\_\_(self, number, balance=0):

    self.number, self.balance = number, balance

  def withdraw(self, amount):

    self.balance -= amount

  def \_\_eq\_\_(self, other):

    print(type(self), type(other))

    return (self.number == other.number) and (type(self) == type(other))

acct = BankAccount(873555333)

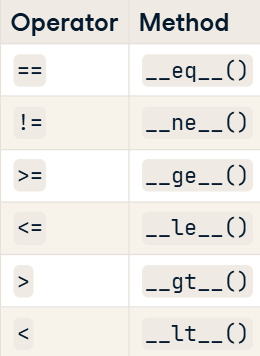
pn = Phone(873555333)

print(acct == pn)

result:



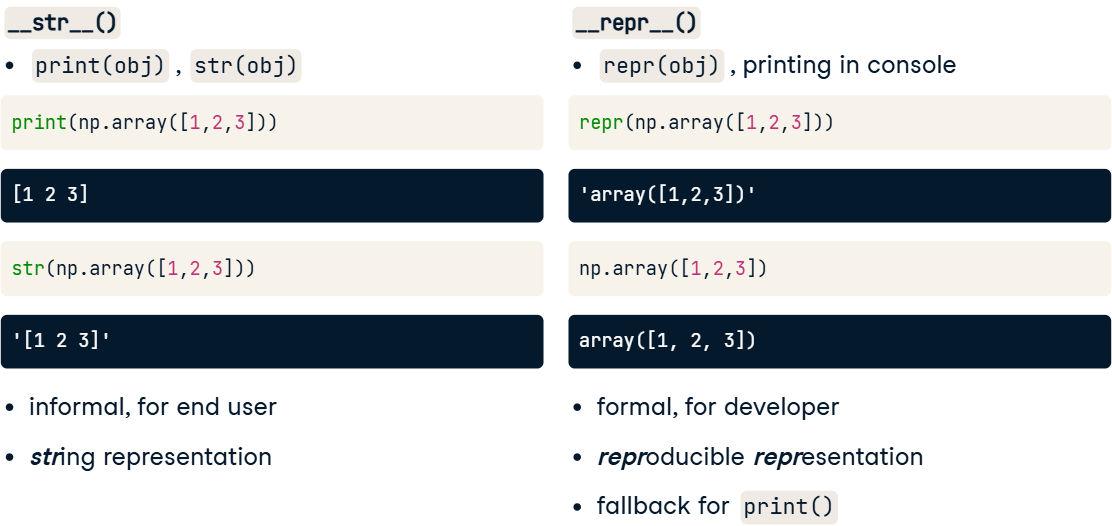
* Python always calls the child's **\_\_eq\_\_()** method when comparing a child object to a parent object.
* Comparison objects (have to be defined within the class as methods):

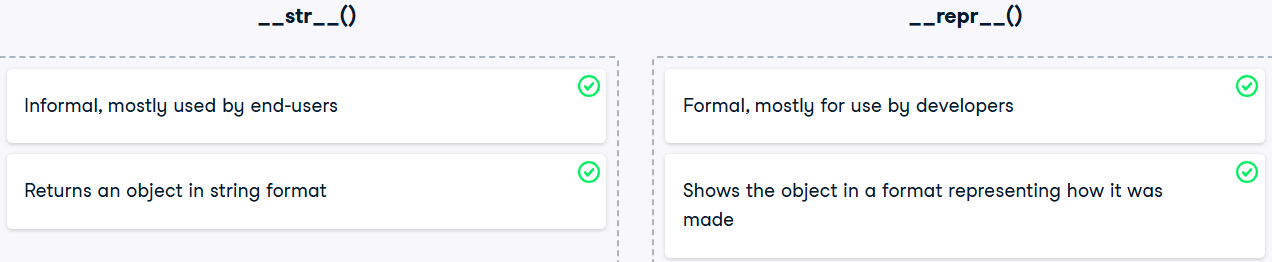


* Also can use ***isinstance()*** for comparison.

#### Inheritance comparison and string representation

* Printing an object of a Class (fallback – запасний варіант):





\_\_str\_\_() is called when you use print() or str() on an object, and \_\_repr\_\_() is called when you use repr() on an object, print the object in the console without calling print(), or instead of \_\_str\_\_() if \_\_str\_\_() is not defined.

* **\_\_repr\_\_** implemenation:

class Employee:

    def \_\_init\_\_(self, name, salary=30000):

        self.name, self.salary = name, salary

    def \_\_repr\_\_(self):

        emp\_str = f"Employee('{self.name}', {self.salary})"

        return emp\_str

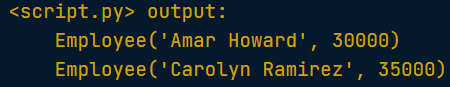
emp1 = Employee("Amar Howard", 30000)

print(repr(emp1))

emp2 = Employee("Carolyn Ramirez", 35000)

print(repr(emp2))

result:



* **\_\_str\_\_** implemenation:

class Employee:

    def \_\_init\_\_(self, name, salary=30000):

        self.name, self.salary = name, salary

    def \_\_str\_\_(self):

      emp\_str = f"""Employee name: {self.name}

    Employee salary: {self.salary}"""

      return emp\_str

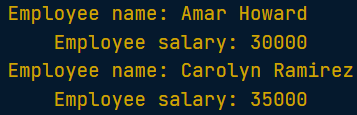
emp1 = Employee("Amar Howard", 30000)

print(emp1)

emp2 = Employee("Carolyn Ramirez", 35000)

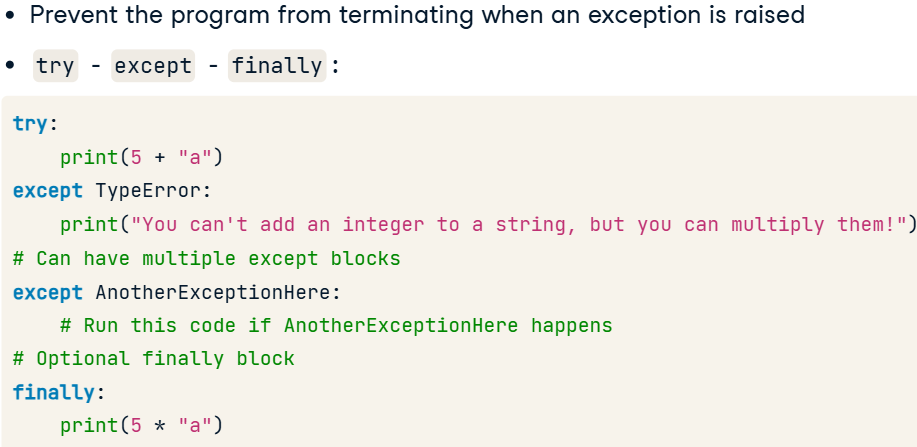
print(emp2)

result:



#### Exceptions

* Catching the error and handling it:



* + Example:

def invert\_at\_index(x, ind):

  try:

    return 1 / x[ind]

  except ZeroDivisionError:

    print("Cannot divide by zero!")

  except IndexError:

    print("Index out of range!")

a\_list = [5, 6, 0, 7]

# Works okay

print(invert\_at\_index(a\_list, 1))

# Potential ZeroDivisionError

print(invert\_at\_index(a\_list, 2))

# Potential IndexError

print(invert\_at\_index(a\_list, 5))

* Raising exceptions and creating our own exceptions:

class SalaryError(ValueError):

  pass

class BonusError(SalaryError):

  pass

class Employee:

  MIN\_SALARY = 30000

  MAX\_BONUS = 5000

  def \_\_init\_\_(self, name, salary = 30000):

    self.name = name

    if salary < Employee.MIN\_SALARY:

      raise SalaryError("Salary is too low!")

**self.salary = salary**

  def give\_bonus(self, amount):

    if amount > Employee.MAX\_BONUS:

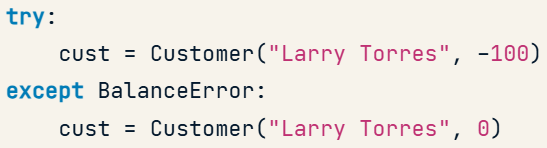
      raise BonusError("The bonus amount is too high!")

    elif (self.salary + amount) < Employee.MIN\_SALARY:

      raise SalaryError("The salary after bonus is too low!")

**self.salary += amount**

* Exeptions are classes, that terminate the programm and don’t create an object.
* Catching custom exceptions:



### Best Practices of Class Design

#### Designing for inheritance and polymorphism

* Liskov substitution principle:

Base class should be interchangeable with any of its subclasses without altering any properties of the program.

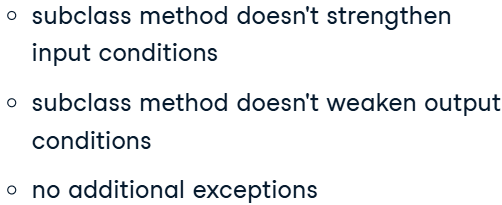
Similar methods should work the same way.

1. **Синтаксически:**
   * Подписи функций должны совпадать: одинаковые параметры и возвращаемые значения.
2. **Семантически:**
   * Логика программы не нарушается: состояние объектов и программы остается корректным.
   * Пример:

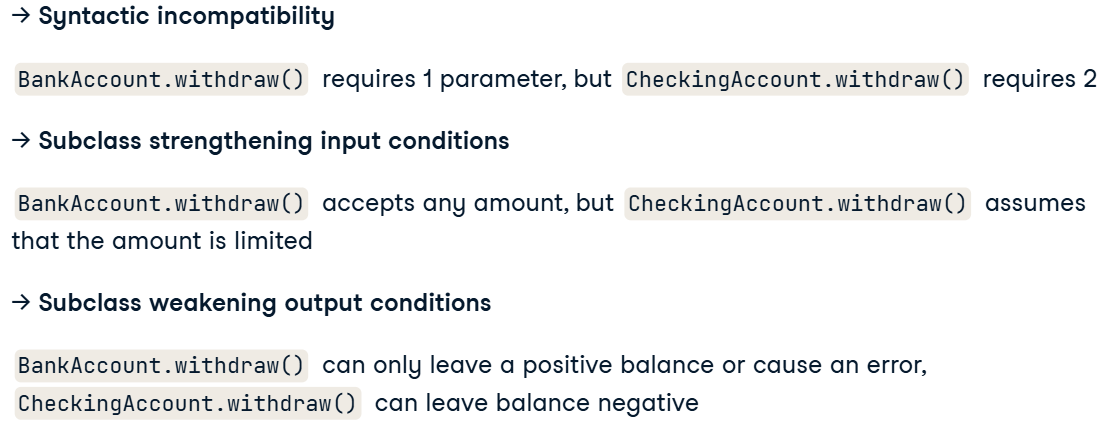
Представь, что есть команда "апорт!", и ты бросаешь палку. Если ты говоришь это лабрадору или его щенку, они оба должны понять команду одинаково: побежать за палкой, принести её и ничего лишнего не сделать. Если щенок вместо палки принесёт тапок — это уже нарушает семантику. Значит, он не совсем заменяет взрослого лабрадора.

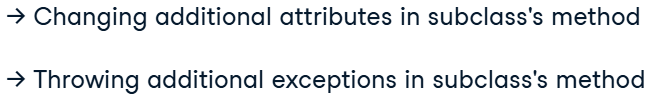
Проще говоря, подкласс должен полностью "вести себя" как базовый класс.

* + In details:



* Violating LSP:



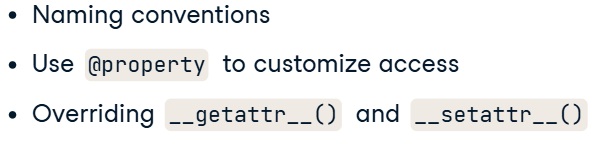


* Ultimate rule:

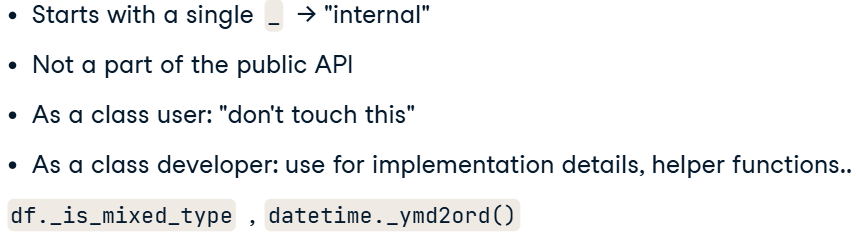


#### Managing data access: private attributes

* Restricting access:



* Naming convention (Internal attributes) – you should not touch it:



* Naming convention (Private attributes):



* + Details:



* + Main things:

1. \_\_attr используется для защиты данных через **перемешивание имен** (например, \_\_attr → \_ClassName\_\_attr).
2. Помогает избежать **конфликтов имен** при наследовании.
3. Предназначено для **внутреннего использования**, но не является настоящей приватностью.
4. Не путать с \_\_magic\_\_ методами, которые зарезервированы Python.
5. Используйте для защиты важных атрибутов и методов от переопределения.

#### Properties

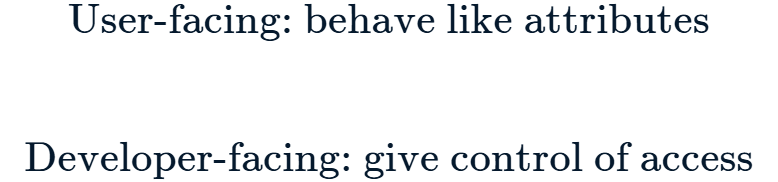
* Working with one *internal* atrribute and it’s modification:



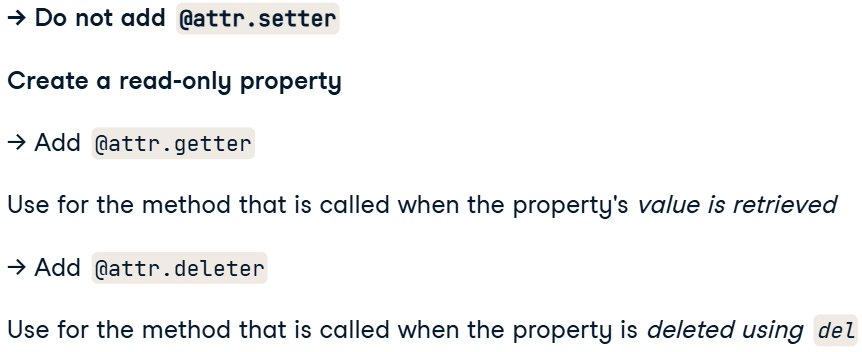
* + How it works:



* Why we should use it:



* Other possibilities:

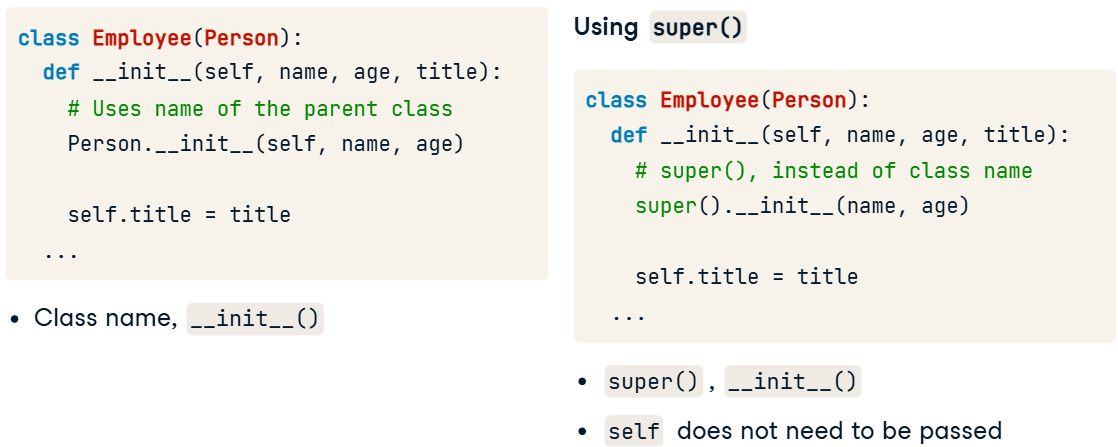


## Intermediate Object-Oriented Programming in Python

### Overloading and Multiple Inheritance

#### Fundamentals of Object-Oriented Programming

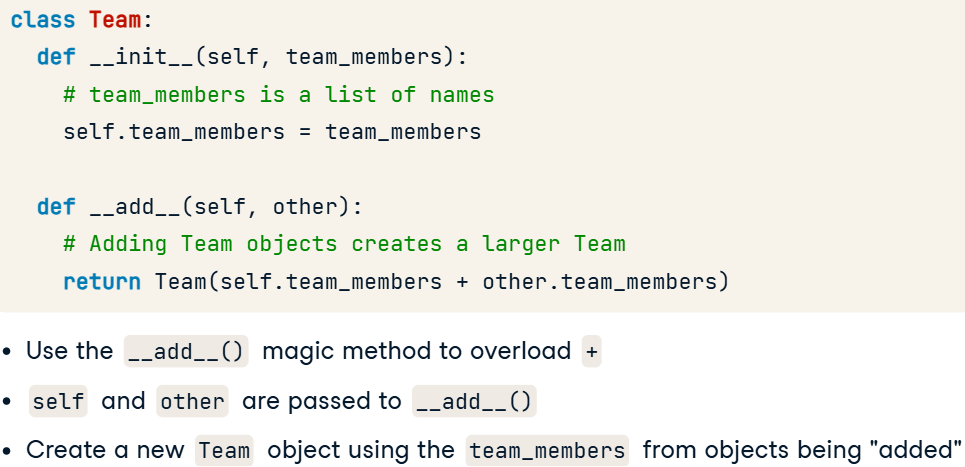
* super():



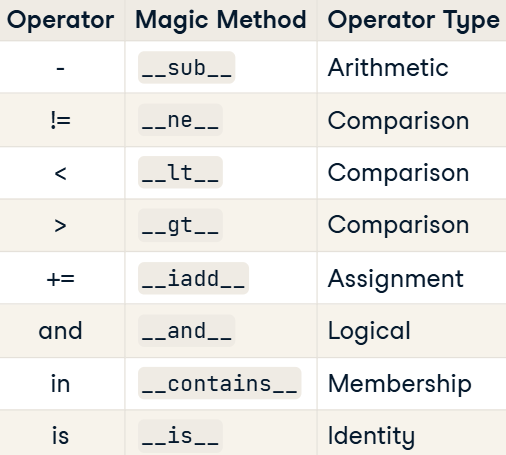
* Overriding – child implements a method that was inherited from parent in a new way.

#### Overloading Python Operators

* Overloading the + operator:



* Other overloading operators:



* Code fpr \_\_add\_\_ overload:

class Storage:

  def \_\_init\_\_(self, capacity):

    self.capacity = capacity

  def \_\_add\_\_(self, other):

    return Storage(self.capacity + other.capacity)

onboard\_storage = Storage(128)

external\_drive = Storage(64)

total\_storage = onboard\_storage + external\_drive

print(total\_storage.capacity)

* Creating a new type of object:

class Network:

  def \_\_init\_\_(self, ip\_addresses):

    self.ip\_addresses = ip\_addresses

class Computer:

  def \_\_init\_\_(self, operating\_system, ip\_address):

    self.operating\_system = operating\_system

    self.ip\_address = ip\_address

  def \_\_add\_\_(self, other):

    if self.operating\_system == other.operating\_system:

      return **Network**([self.ip\_address, other.ip\_address])

    raise Exception("Incompatible operating systems.")

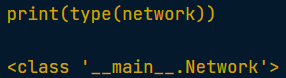
morgans\_computer = Computer("Windows", "182.112.81.991")

jennys\_computer = Computer("Windows", "177.511.64.162")

network = morgans\_computer + jennys\_computer

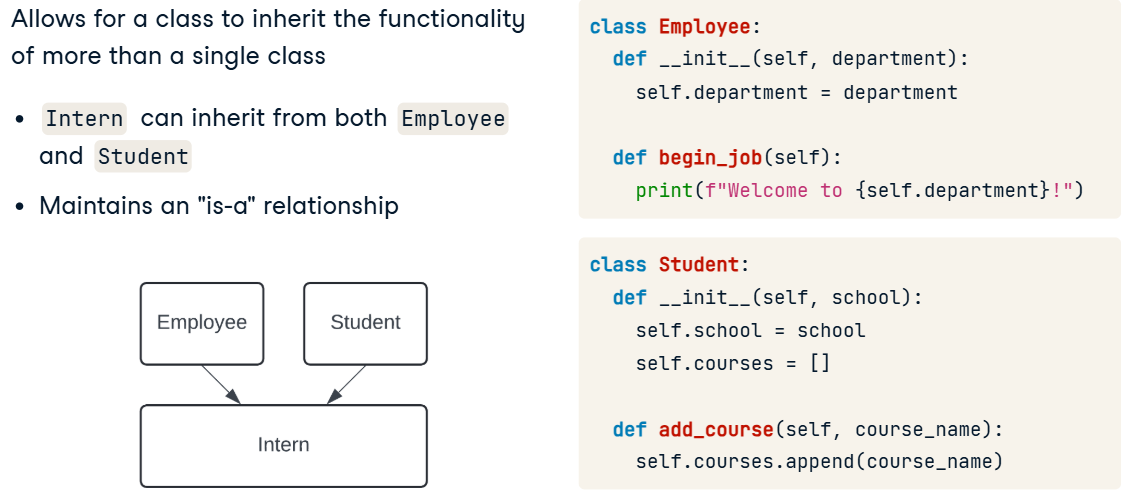
print(type(network))

result:

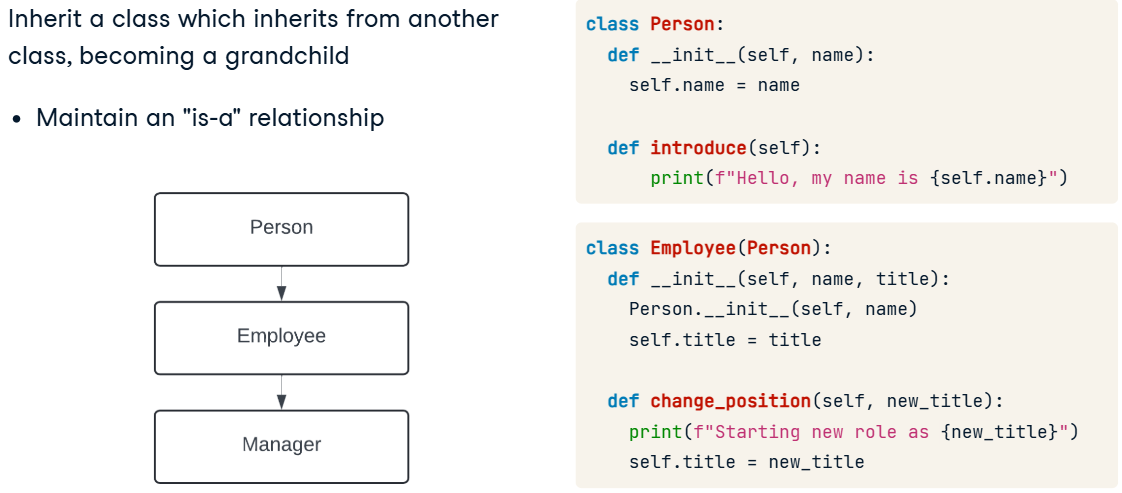


#### Multiple Inheritance

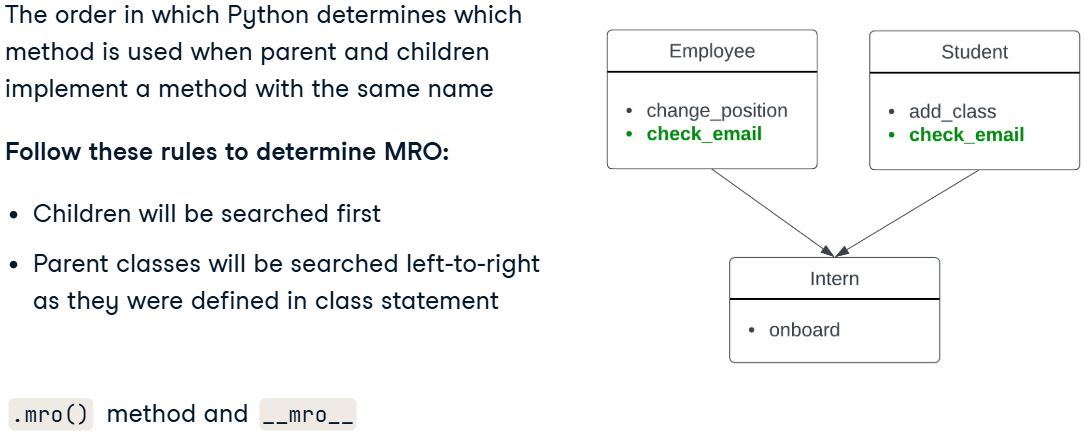
* *Multiple inheritance* – allows a class to inherit the functionality of more than one class.



* *Multilevel inheritance* (another form of multiple inheritance) – inherit a class which inherits from another class becoming a *grandchild*.
  + Process:



* Method resolution order (MRO):



* + class.mro() – the order in which Python will search for methods.

Smartphone.mro()

result:



* Multiple inheritance:

class Smartphone(Computer, Telephone):

  def \_\_init\_\_(self, brand, phone\_number, music\_app):

    Computer.\_\_init\_\_(self, brand)

    Telephone.\_\_init\_\_(self, phone\_number)

    self.music\_app = music\_app

  def play\_music(self, song):

    print(f"Playing {song} using {self.music\_app}")

personal\_phone = Smartphone("Macrosung", "801-932-7629", "Dotify")

# Browse the internet, make a call to Alex, and play music

personal\_phone.browse\_internet()

personal\_phone.make\_call("Alex")

personal\_phone.play\_music("Creeks and Highways")

* Multilevel inheritance:

class Computer:

  def \_\_init\_\_(self, brand):

    self.brand = brand

  def browse\_internet(self):

    print(f"Using {self.brand}'s default internet browser.")

class Tablet(*Computer*):

  def \_\_init\_\_(self, brand, apps):

    Computer.\_\_init\_\_(self, brand)

    self.apps = apps

  def uninstall\_app(self, app):

    if app in self.apps:

      self.apps.remove(app)

class Smartphone(*Tablet*):

  def \_\_init\_\_(self, brand, apps, phone\_number):

    Tablet.\_\_init\_\_(self, brand, apps)

    self.phone\_number = phone\_number

  def send\_text(self, message, recipient):

    print(f"Sending {message} to {recipient} from {self.phone\_number}")

personal\_phone = Smartphone("Macrosung", ["Weather", "Camera"], "801-932-7629")

personal\_phone.browse\_internet()

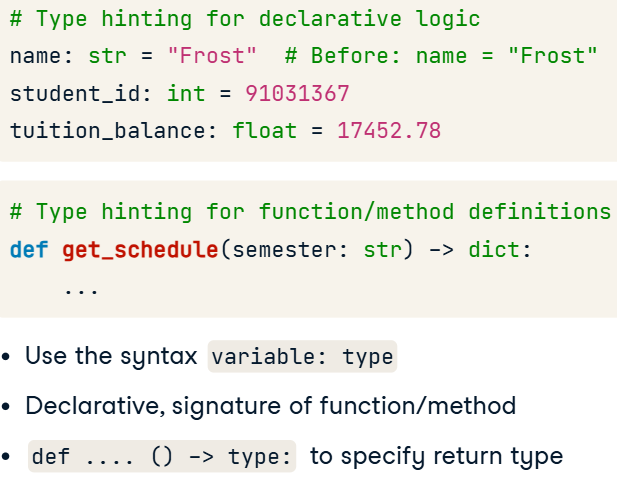
personal\_phone.uninstall\_app("Weather")

personal\_phone.send\_text("Time for a new mission!", "Chuck")

### Custom Class Features and Type Hints

#### Type Hints

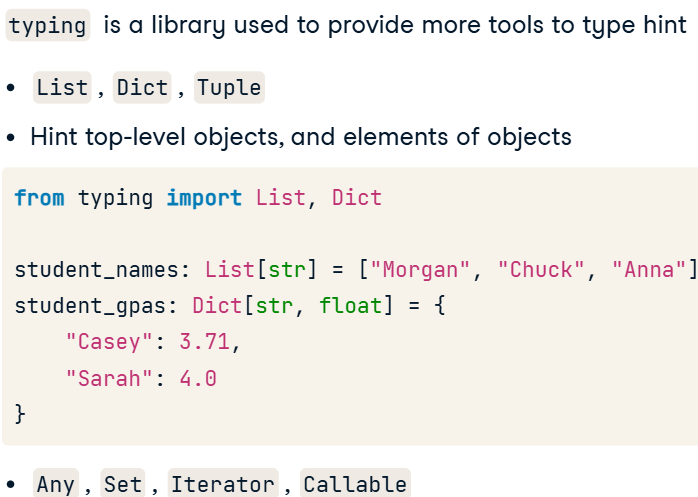
* Allow for information about the type of an object to be added to code.
  + Creating type hints params:



* Why we should use them:

1. **Читаемость** — понятнее, какие типы данных ожидаются.
2. **Меньше ошибок** — статический анализ помогает находить проблемы.
3. **Документация** — заменяют часть комментариев.
4. **Удобство** — автодополнение и подсказки в IDE.

* ***Typing*** library:



* + Code:

class Agent:

  def \_\_init\_\_(self, codename: str, missions: int):

    self.codename: str = codename

    self.missions: int = missions

  def add\_mission(self, location: str) -> None:

    self.missions += 1

    print(f"{self.codename} completed a mission in " + \

          f"{location}. This was mission #{self.missions}")

chuck: Agent = Agent("Charles Carmichael", 37)

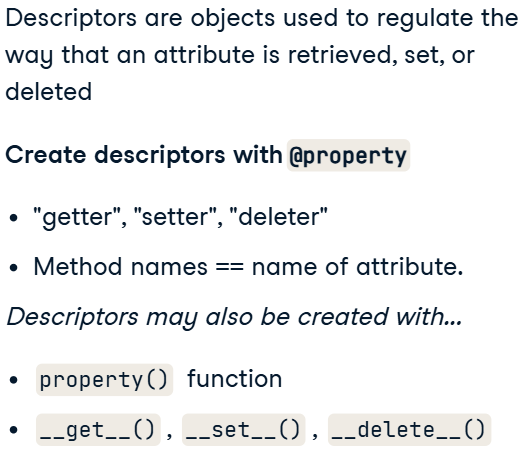
locations: List[str] = ["Burbank", "Paris", "Prague"]

for location in locations:

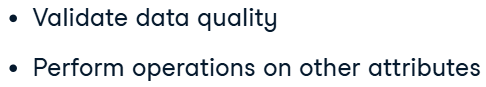
  chuck.add\_mission(location)

#### Descriptors

* ***Descriptors***are used to regulate the way that an attribute is retrieved, set or deleted.
  + Additional info:



* Descriptors:
  + **@property** – contols how the attribute is retrived.
  + **@attr.setter** – contols how the attribute is stored:



* + **@attr.deleter** – controls the attr deletion process.
* Code:

class BankAccount:

  def \_\_init\_\_(self, email):

    self.\_email = email

  @property

  def email(self):

    return f"Email for this account is: {self.\_email}"

  @email.setter

  def email(self, new\_email\_address):

    if "@" in new\_email\_address:

      self.\_email = new\_email\_address

    else:

      print("Please make sure to enter a valid email.")

  @email.deleter

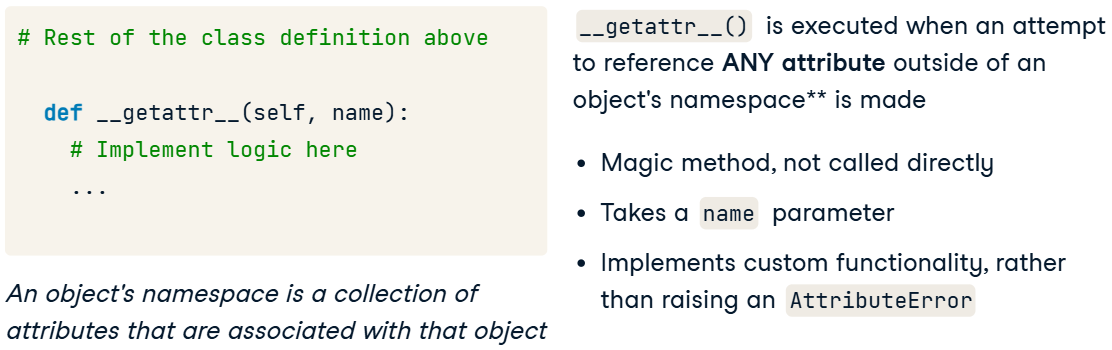
  def email(self):

    del self.\_email

    print("Email deleted, make sure to add a new email!")

#### Customizing Attribute Access

* \_\_getattr\_\_:



* + Code:

class BankAccount:

  def \_\_init\_\_(self, account\_number):

    self.account\_number = account\_number

  def \_\_getattr\_\_(self, name):

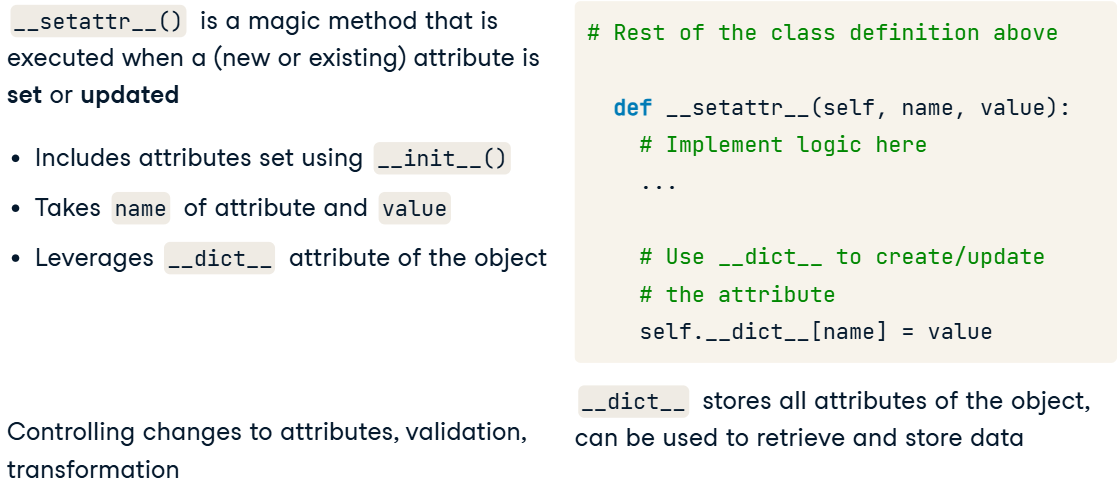
    print(f"""{name} is not defined in BankAccount object.

      Please define this attribute if needed.""")

checking\_account = BankAccount("123456")

checking\_account.routing\_number

* \_\_setattr\_\_ – is used to customize the attributes behaviour:



* + Code:

class BankAccount:

  def \_\_init\_\_(self, account\_number):

    self.account\_number = account\_number

  def \_\_setattr\_\_(self, name, value):

    if name in ["account\_number", "balance"]:

      print(f"{name} is an allowed attribute.")

      self.\_\_dict\_\_[name] = value

    else:

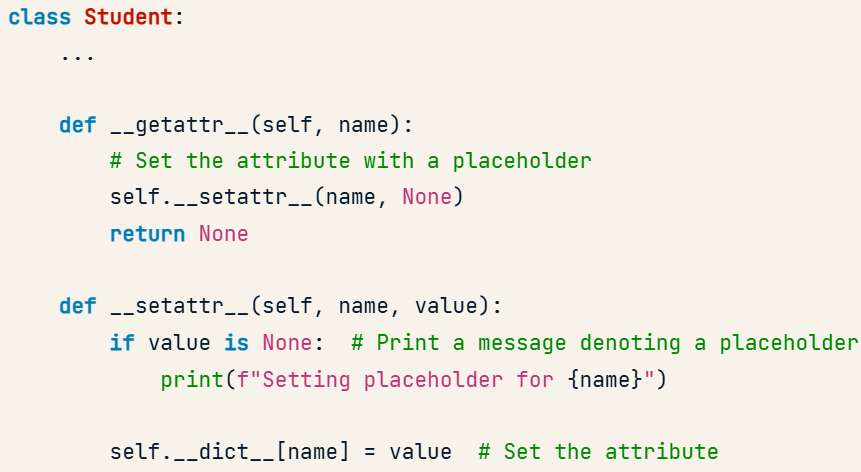
      print(f"Invalid Attribute: {name}")

savings\_account = BankAccount("12345678")

savings\_account.balance = 100

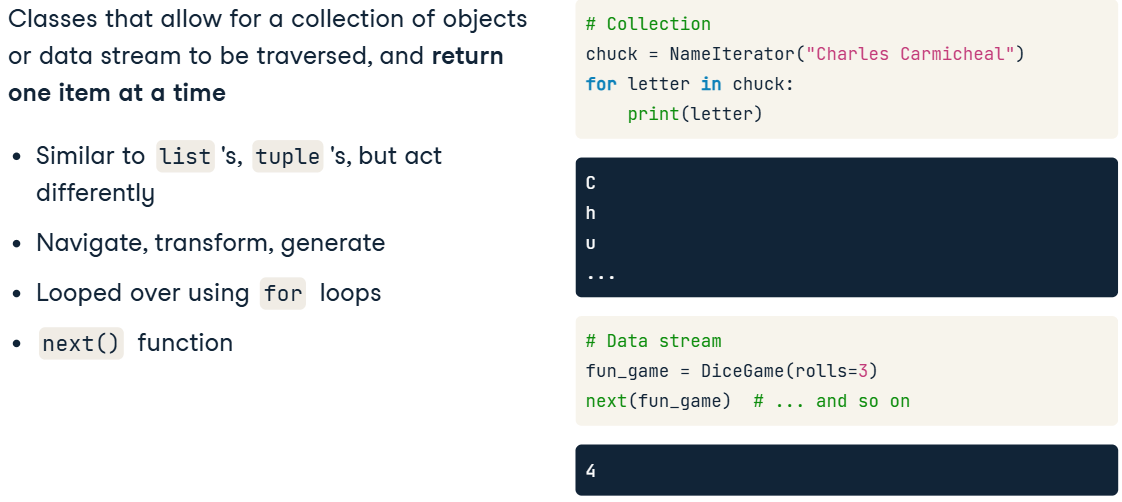
savings\_account.beneficiary = "Anna Wu"

* \_\_setattr\_\_ and \_\_getattr\_\_ together:

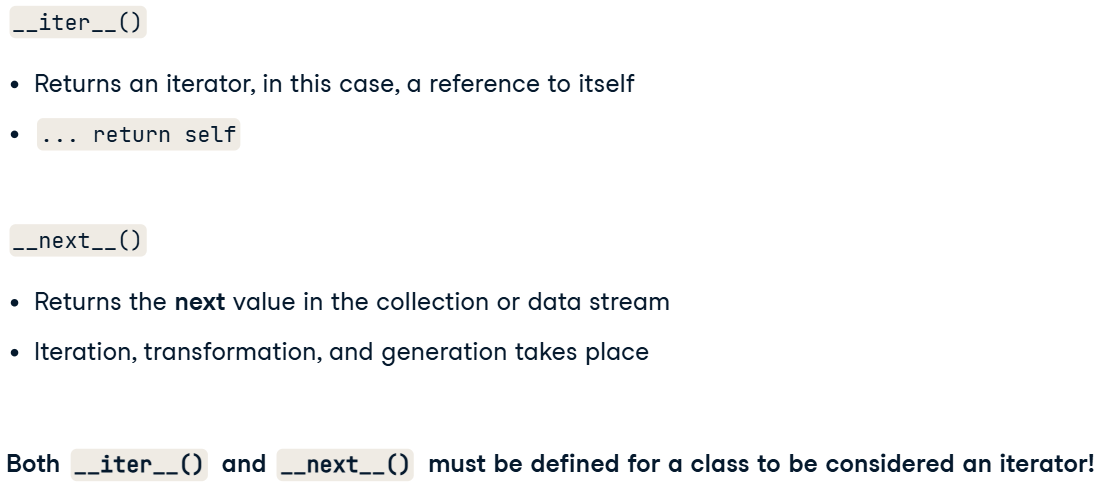


#### Custom Iterators

* Iterators:



* Iterator protocol:



* + Code (+ stopping infinite iteration):

class Playlist:

  def \_\_init\_\_(self, songs, shuffle=False):

    self.songs = songs

    self.index = 0

    if shuffle:

      random.shuffle(self.songs)

  def \_\_iter\_\_(self):

    return self

  def \_\_next\_\_(self):

    if self.index >= len(self.songs): # stop infinite iteration

      raise StopIteration

    song = self.songs[self.index]

    self.index += 1

    return song

* Handling *StopIteration* exception:

songs = ["Hooked on a Feeling", "Yesterday", "Mr. Blue Sky"]

classic\_rock\_playlist = Playlist(songs, shuffle=True)

while True:

    try:

        print(next(classic\_rock\_playlist))

    except StopIteration:

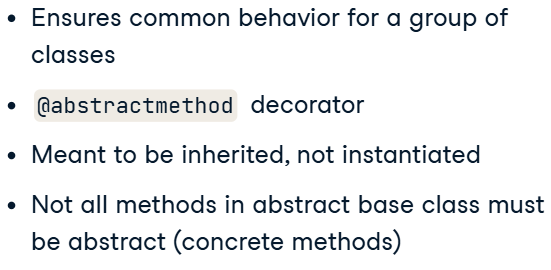
        print("Reached end of playlist!")

        break

### Object-oriented design patterns

#### Abstract Base Classes

* Abstract Base Class – creates a *blueprint* for other classes (that are quite similar) by defining *abstaract methods* that must be implemented by all children (classes which inherit the ABC).
  + How it works:

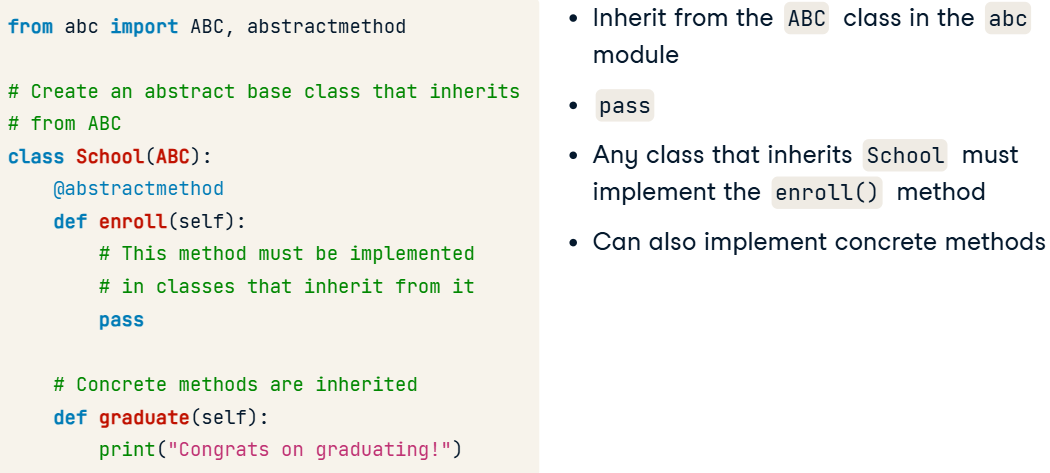


* + - Addition info:

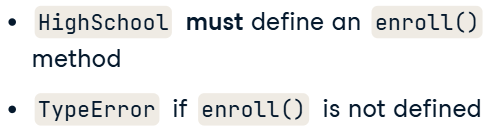
*Нельзя создавать экземпляры ABC* → Это значит, что объект абстрактного класса создать нельзя. Его можно только унаследовать и реализовать в дочернем классе.

*Может содержать как абстрактные, так и конкретные* методы → В абстрактном классе могут быть как методы без реализации (@abstractmethod), так и обычные методы с реализацией, которые можно использовать в дочерних классах.

* Creating ABC:



* + Implementing ABC:



* Code for ABC + Inheriting:

from abc import ABC, abstractmethod

class Company(ABC):

  @abstractmethod

  def pay\_taxes(self):

    pass

  def report\_revenue(self): # concrete method

    print(f"{self.name} is reporting ${self.revenue} of revenue")

class Manufacturing(Company):

  def \_\_init\_\_(self, name, revenue):

    self.name = name

    self.revenue = revenue

  def pay\_taxes(self, tax\_rate):

    tax\_amount = self.revenue \* tax\_rate

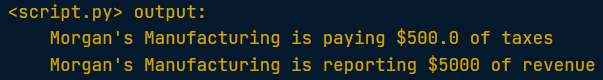
    print(f"{self.name} is paying ${tax\_amount} of taxes")

m = Manufacturing("Morgan's Manufacturing", 5000)

m.pay\_taxes(0.1)

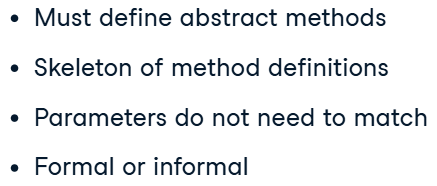
m.report\_revenue()

result:



#### Interfaces

* Interfaces – is a class that implements only abstract methods and create *contracts* (домовленість) with classes which implement the interface.
  + How it works:



* + - Abstract methods provide a skeleton of a method definition, only ***including***: the method name, parameters, and pass keyword.
    - Parameters in an abstract method typically act as a guideline for all classes implementing the interface, but don't need to match exactly.
* *Formal interfaces* defined like ABC using the ABC class and AM decorator:

class Business(ABC):

  @abstractmethod

  def sell\_product(self, product\_name, price, quantity):

    pass

class Bakery(Business):

  def \_\_init\_\_(self, business\_name):

    self.business\_name = business\_name

  def sell\_product(self, product\_name, price, quantity):

    total\_revenue = price \* quantity

    print(f"""{self.business\_name} sold {quantity}

          {product\_name} for a total of ${total\_revenue}""")

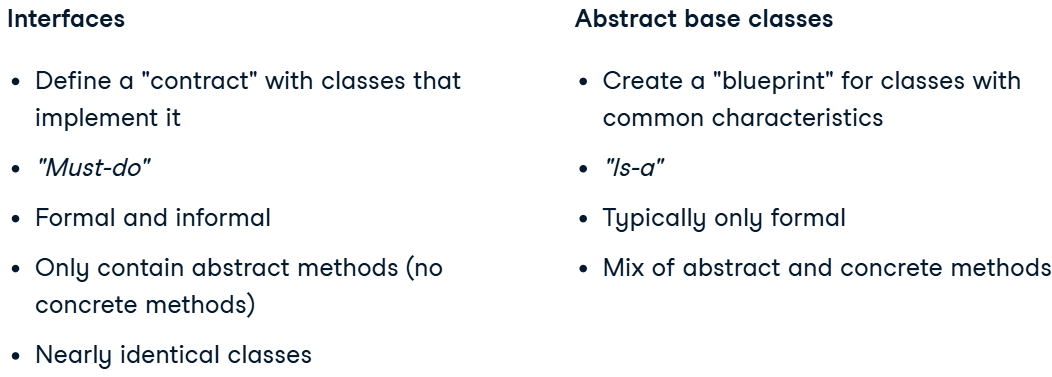
try:

  blue\_eyed\_baker = Bakery("Blue Eyed Baker")

except Exception as e:

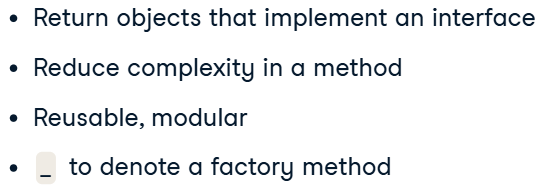
  print(e)

* Interfaces vs ABC:



#### Factory methods

* Factory method design pattern uses *factory methods* to create objects that will be used in another method. Must return objects which implement an *interface*.
  + How it works:



* Creating a factory method:
  + Customer – a product;
  + RewardsMember, NewCusomer – concrete products of the Customer interface.

class Customer(ABC):

  @abstractmethod

  def make\_payment(self, price):

    pass

class RewardsMember(Customer):

  def make\_payment(self, price):

    print(f"""Total price for rewards member is

          ${price \* .90}, which is 10% off""")

class NewCustomer(Customer):

  def make\_payment(self, price):

    print(f"""Total price for new customer is ${price}""")

class Checkout:

  # factory method

  def \_get\_customer(self, customer\_type):

    if customer\_type == "Rewards Member":

      return RewardsMember()

    elif customer\_type == "New Customer":

      return NewCustomer()

  def complete\_transaction(self, customer\_type, price):

    customer = self.\_get\_customer(customer\_type)

    return customer.make\_payment(price)

* Application:
  + SQL:

items\_pipeline = DataPipeline()

items\_pipeline.extract\_data("Redshift", "SELECT \* FROM items;")

items\_pipeline.extract\_data("Postgres", "SELECT \* FROM items;")

etl\_pipeline = DataPipeline()

etl\_pipeline.extract\_data("Redshift", "SELECT \* FROM sales;")

* + Bots:

class LLM(ABC):

  @abstractmethod

  def complete\_sentence(self, prompt):

    pass

class OpenAI(LLM):

  def complete\_sentence(self, prompt):

    return prompt + " ... OpenAI end of sentence."

class Anthropic(LLM):

  def complete\_sentence(self, prompt):

    return prompt + " ... Anthropic end of sentence."

class ChatBot:

  def \_get\_llm(self, provider):

    if provider == "OpenAI":

      return OpenAI()

    elif provider == "Anthropic":

      return Anthropic()

  def chat(self, provider, prompt):

    llm = self.\_get\_llm(provider)

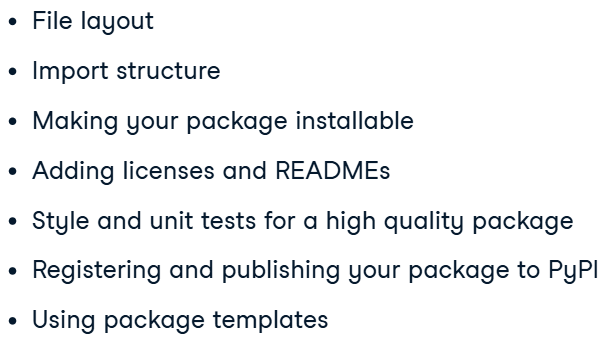
    return llm.complete\_sentence(prompt)

## Developing Python Packages

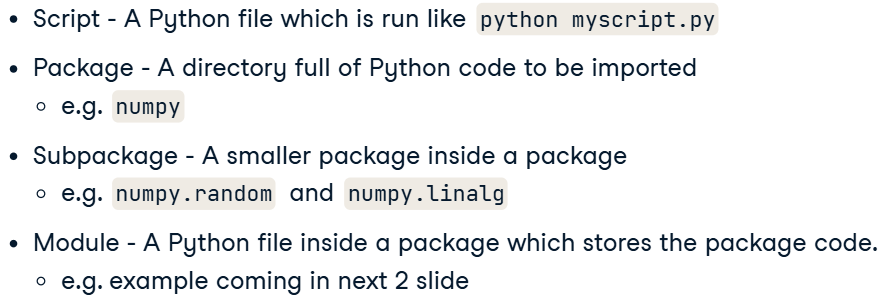
### From Loose Code to Local Package

#### Starting a package

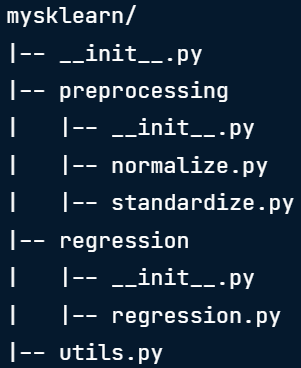
* Course contents:



* Terms (script, package, subpackage, module):



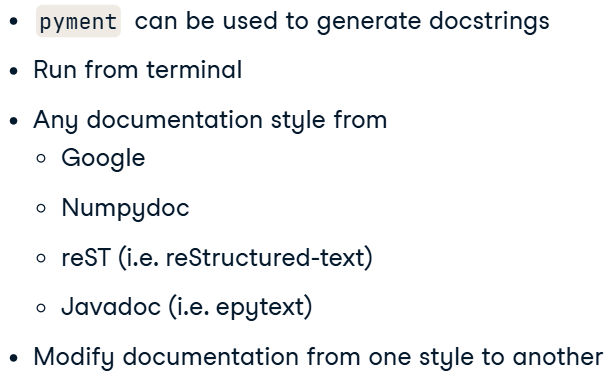
* Subpackages structure (to organise your code):



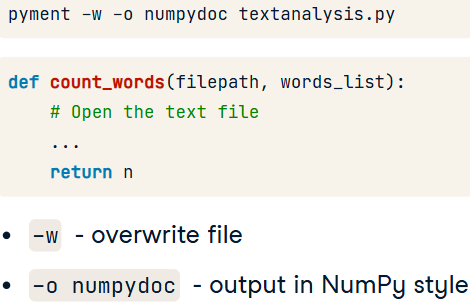
* Structure: You should place closely related functions and classes in the same module, and related modules in the same subpackage.

#### Documentation

* ***Pyment*** (can also modify docs):



* + Code:



* Summarizing your package (in \_\_init\_\_.py):
  + documentation in package:

"""

impyrial

========

A package for converting between imperial

measurements of length and weight.

"""

* + documentation in subpackage:

"""

impyrial.length

===============

Length conversion between imperial units.

"""

* + Multilevel documentation of function (module):

"""Conversions between inches and larger imperial length units"""

def inches\_to\_feet(x, reverse=False):

"""Convert lengths between inches and feet.

...

* Doc for numpy:

def inches\_to\_feet(x, reverse=False):

"""Convert lengths between inches and feet.

Parameters

----------

x : numpy.ndarray

Lengths in feet.

reverse : bool, optional

If true this function converts from feet to inches instead of the default behavior of inches to feet. (Default value = False).

Returns

-------

numpy.ndarray

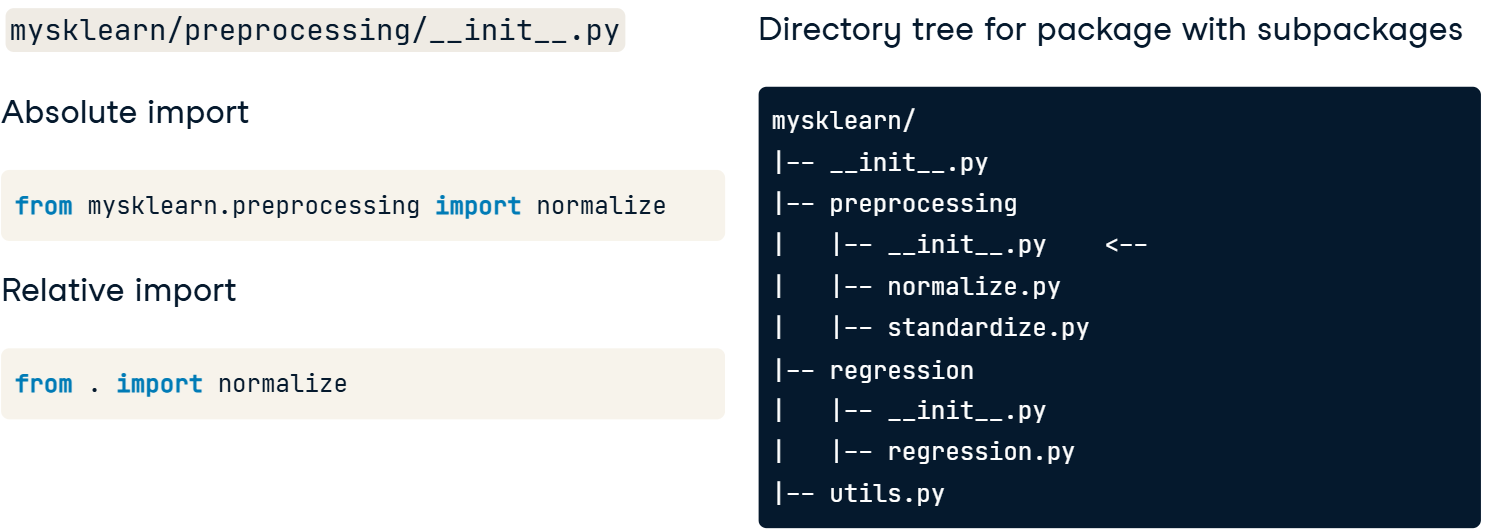
"""

#### Structuring imports

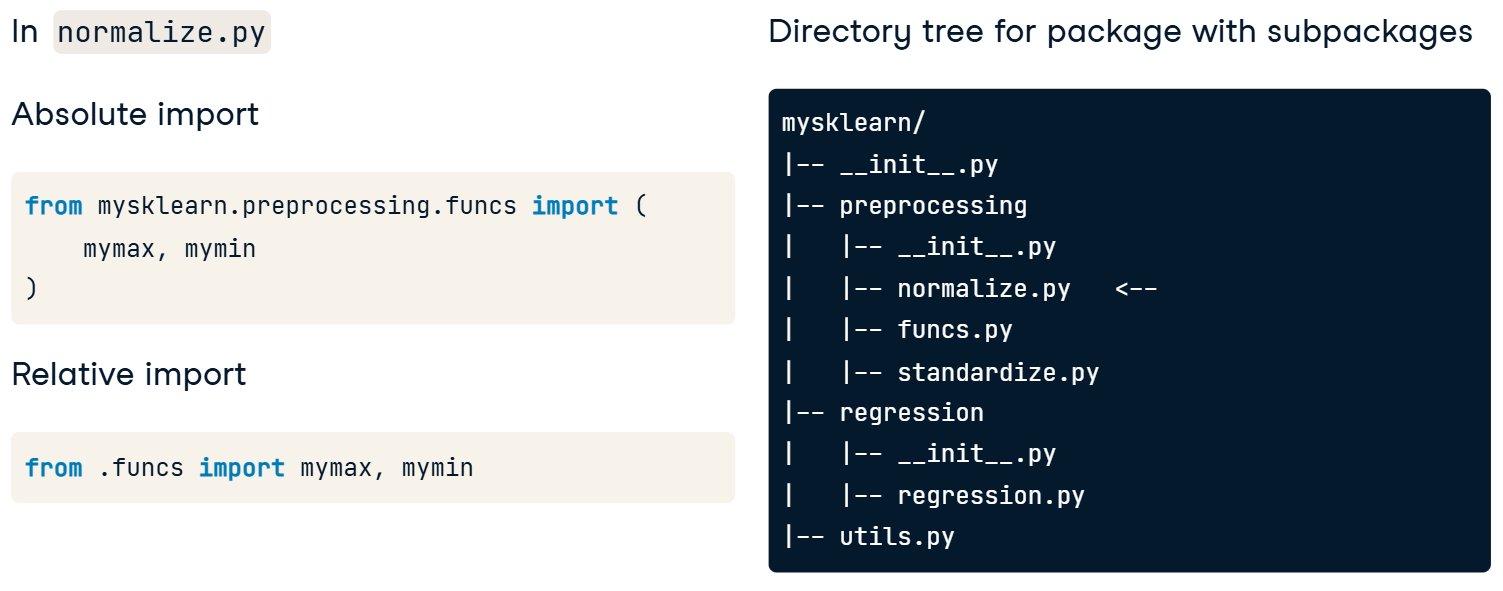
* Each package don’t about subpackages in a directory – so you have to import directly.



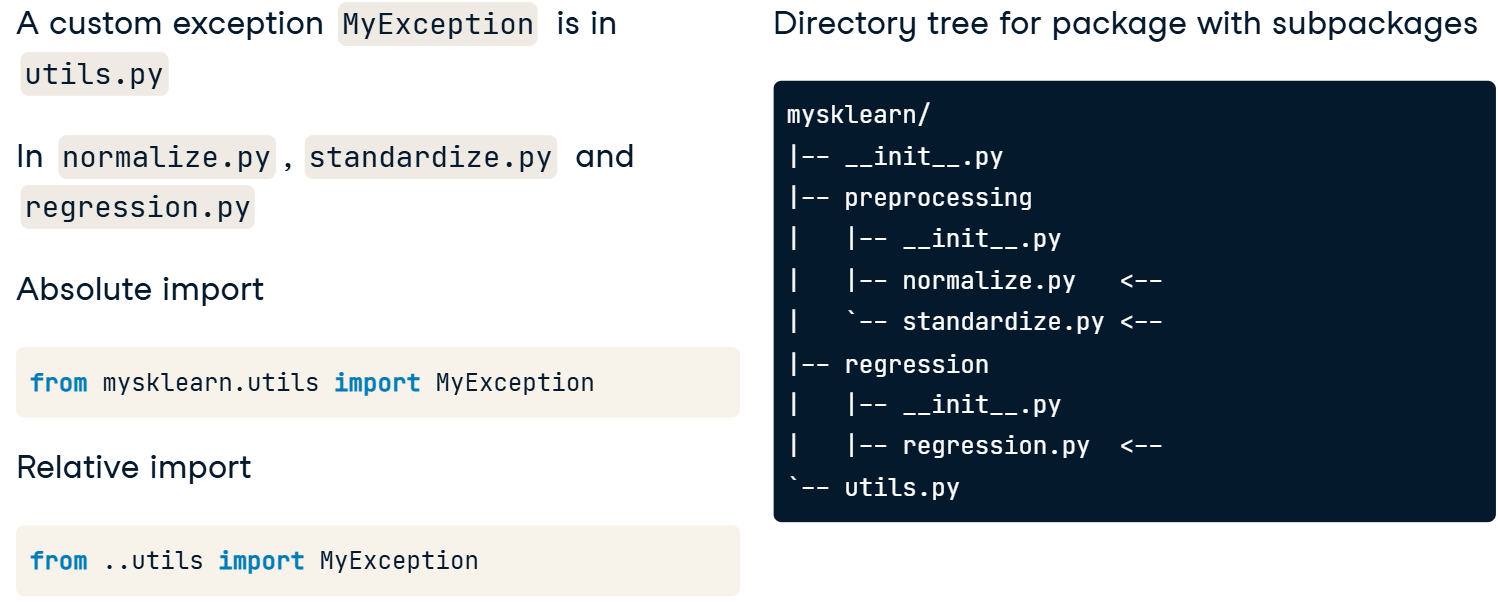
* Importing fonction into subpackages:
  + says: from the current directory import mormalize



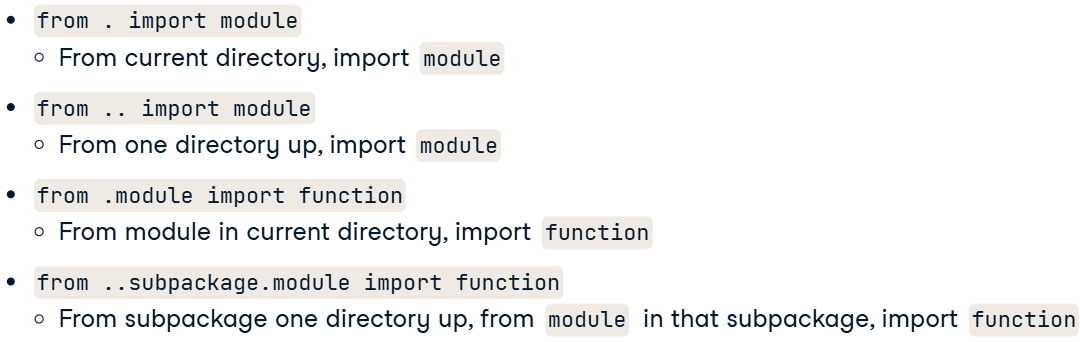
* Importing between sibling modules:



* Importing between modules:



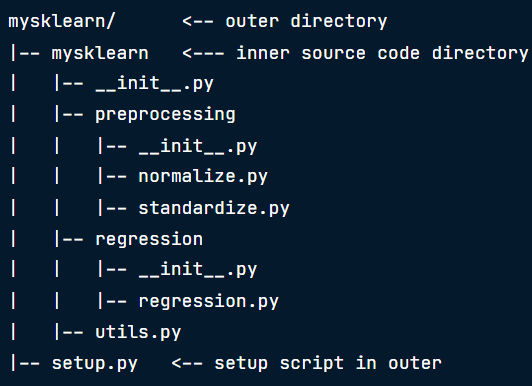
* + . – means parent directory;
  + .. – means parent of the parent directory.
* Cheat Sheet:



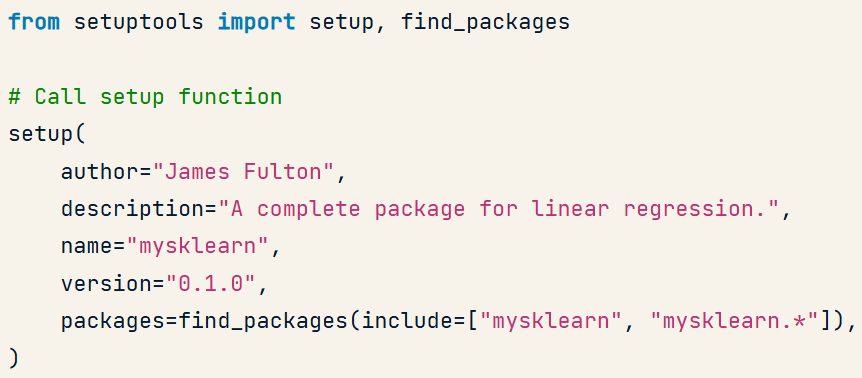
### Install Your Package from Anywhere

#### Installing your own package

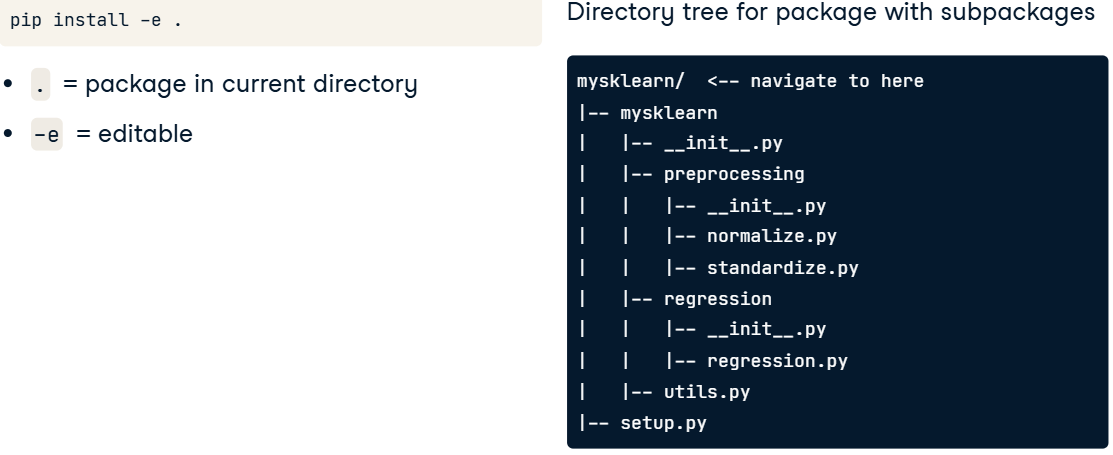
* Naming package and directory:



* Setup.py:



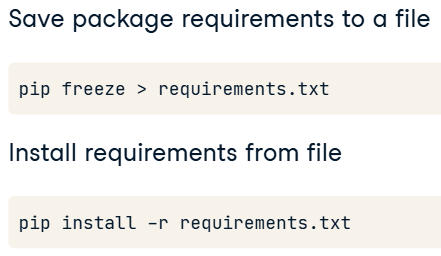
* + packages – incudes package and subpackages.
* Installing a package:



* Controlling dependency:



* Requirements installing (setting):



#### Including licences and writing READMEs

* Writing README:

**#impyrial**

A package for converting between imperial unit lengths and weights.

This package was created for the [DataCamp](https://www.datacamp.com) course "Developing Python Packages".

**### Features**

- Convert lengths between miles, yards, feet and inches.

- Convert weights between hundredweight, stone, pounds and ounces.

**### Usage**

(for coding)

```

import impyrial

# Convert 500 miles to feet

impyrial.length.convert\_unit(500, from\_unit='yd', to\_unit='ft') # returns 1500.0

# Convert 100 ounces to pounds

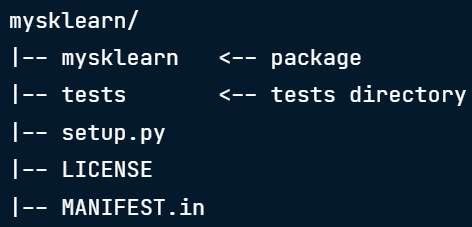
impyrial.weight.convert\_unit(100, from\_unit='oz', to\_unit='lb') # returns 6.25

```

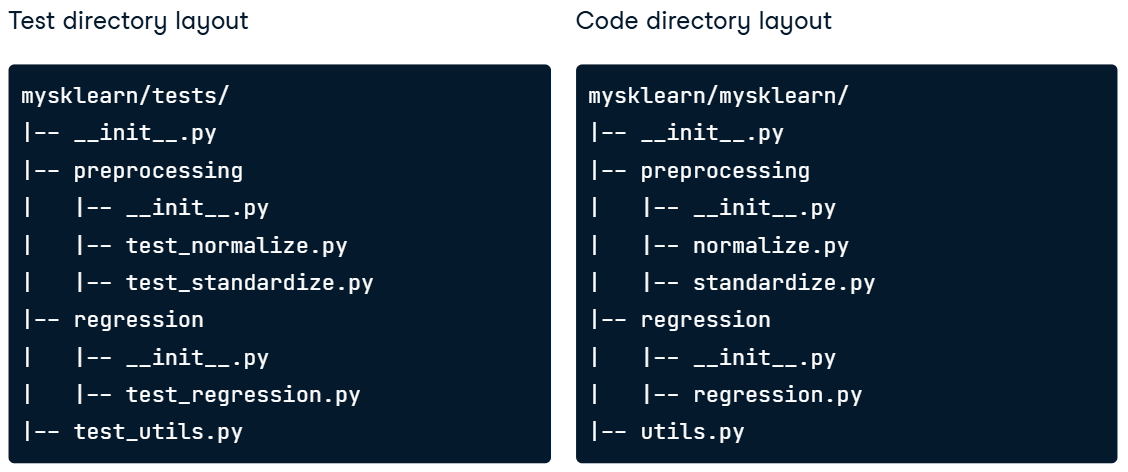
### Increasing Your Package Quality

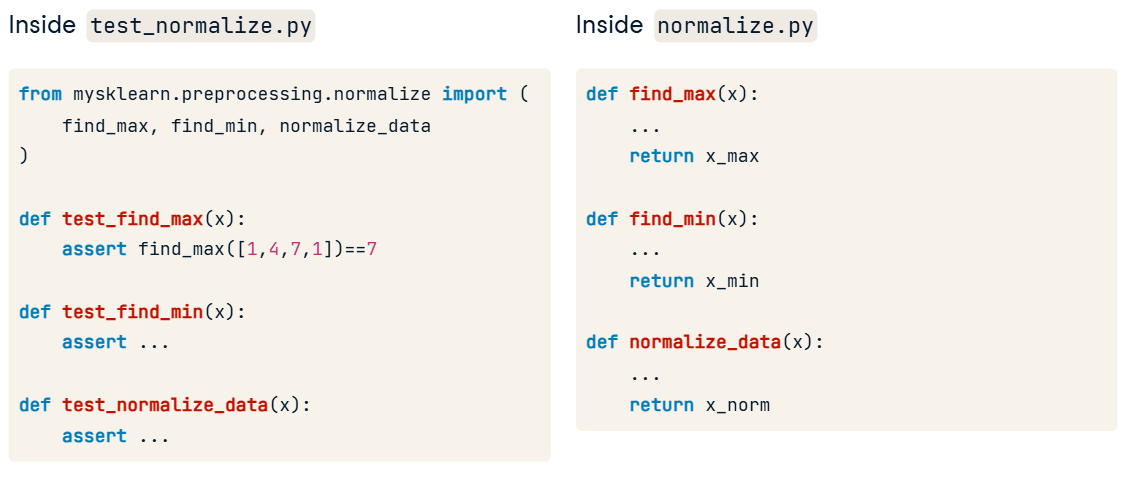
#### Testing your package

* Structuring your tests:



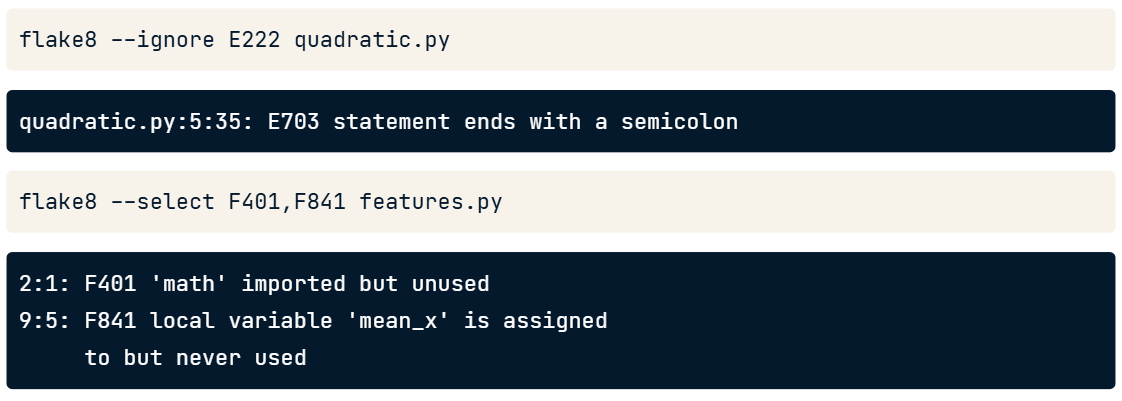
* Organizing tests:



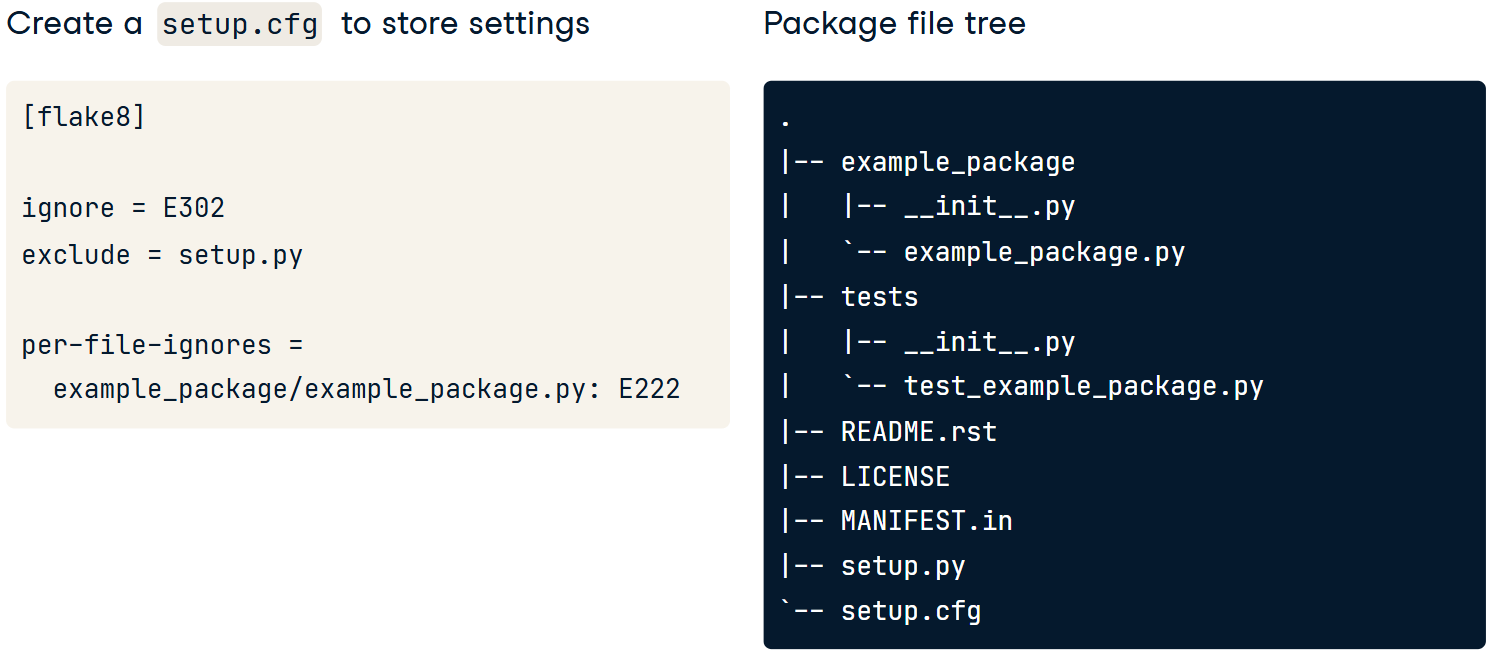
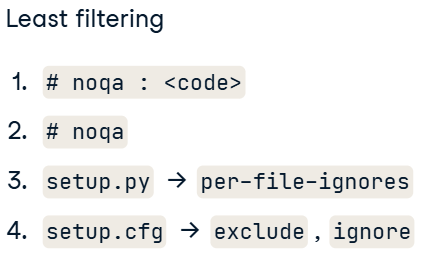


#### Increasing Your Package Quality

* Manipulating the output of violations (flake8):

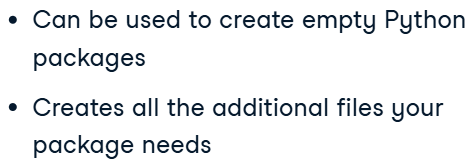


* Filtering:



### Rapid Package Development

* cookiecutter:



* Code:



<https://github.com/audreyfeldroy/cookiecutter-pypackage.git>

## Introduction to Testing in Python

### Creating Tests with pytest

#### Introduction to Testing in Python

* Testing:

def multiple\_of\_two(num):

    if num == 0:

        raise(ValueError)

    return num % 2 == 0

def test\_numbers():

    assert multiple\_of\_two(2) is True

    assert multiple\_of\_two(3) is False

* Exception raises:

def multiple\_of\_two(num):

    if num == 0:

        raise(ValueError)

    return num % 2 == 0

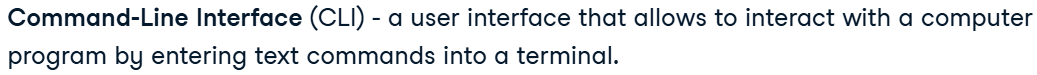
def test\_zero():

  with pytest.raises(ValueError):

    multiple\_of\_two(0)

#### Invoking pytest from CLI

* CLI:



* Filtering functions to test:

