Software Engineering For Data Science (SEDS)

Class: 2 Year 2nd Cycle
Branch: AIDS

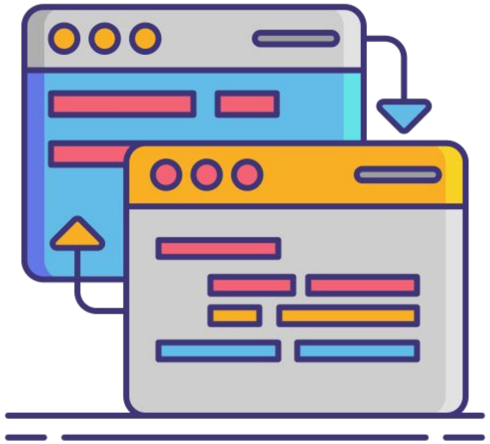
Dr. Belkacem KHALDI| ESI-SBA

Lecture 03:

Advanced Concepts for Python Software Engineering: Modularity, Readability, and Refactoring

Advanced Concepts for Python Software Engineering: Modularity, Readability, and Refactoring

1. Modularity
2. Readability
3. Refactoring



Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Modularity

Modularity

Why Adopting Modularity?

- **Non-modular code** can take the form of **long**, **complicated**, **hard** to read scripts and functions.

- **Modular Code** ⇒ Code divided into shorter functional units.

- **Modular code** ⇒ Code becomes more **readable** and **easier** to **fix** when something breaks.

- **Modular code** ⇒ Provides code **portability** (save time by avoiding re-solving problems you've already solved in a previous project).

Modularity in Python

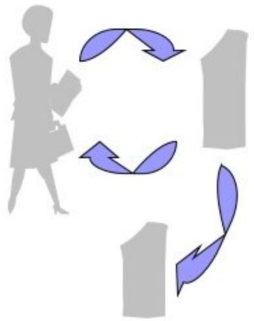
- 3 ways that you can write modular code with Python: **classes**, **modules**, and **packages**

```python
# Import the pandas PACKAGE
import pandas as pd
# Import the sqrt pre-built METHOD from math MODULE
from math import sqrt

# Create some example data
data = {'x': [1.5, 2.25, 3.75, 4.0],
        'y': [sqrt(1.5), sqrt(2.25), sqrt(3.75), sqrt(4.0)]}

# Create a dataframe CLASS object
df = pd.DataFrame(data)

# Use the plot METHOD
df.plot('x', 'y')
```

Modularity: Object-Oriented Programming

Procedural Programming

- Code as a sequence of steps.
- A program is divided into small functions
- Importance is not given to data but to procedure.

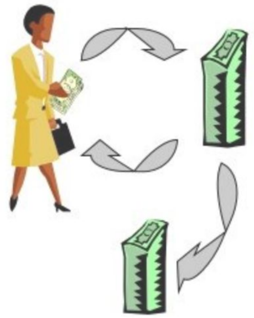


Withdraw, deposit, transfer

- **Why PP?**
  - A good choice for **general-purpose** programming.
  - Offers a **simple**, **intuitive**, and **straightforward** way of writing **sequential code**.
  - **Easier** for **compilers** and **interpreters**.

Object-Oriented Programming

- Code as interactions of objects.
- A program is divided into objects.
- Importance is given to data rather than functions



Customer, money, account

- **Why OOP?**
  - **Organize** your code better.
  - Offre a way of **Securing** Data (encapsulation**)**.
  - Make the code more **reusable** and **maintainable**.
  - **Easier** to **customize** functionality from libraries.

Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Modularity: Object-Oriented Programming

Motivational Example: Pytorch

- Probably you will work a lot with **Pytorch** Deep Learning framework.
- **TinyModel** ⇒ A new class is defined as a subclass of **torch.nn.Module**
- **__init__ ()** ⇒ A method to initialize an instance object constructed from this class.
  - Two linear layers are declared and activation functions.
- **forward()** ⇒ A function describing how the network is connected is declared in function.
- **tinymodel** ⇒ An object is instantiated from the **TinyModel** class.

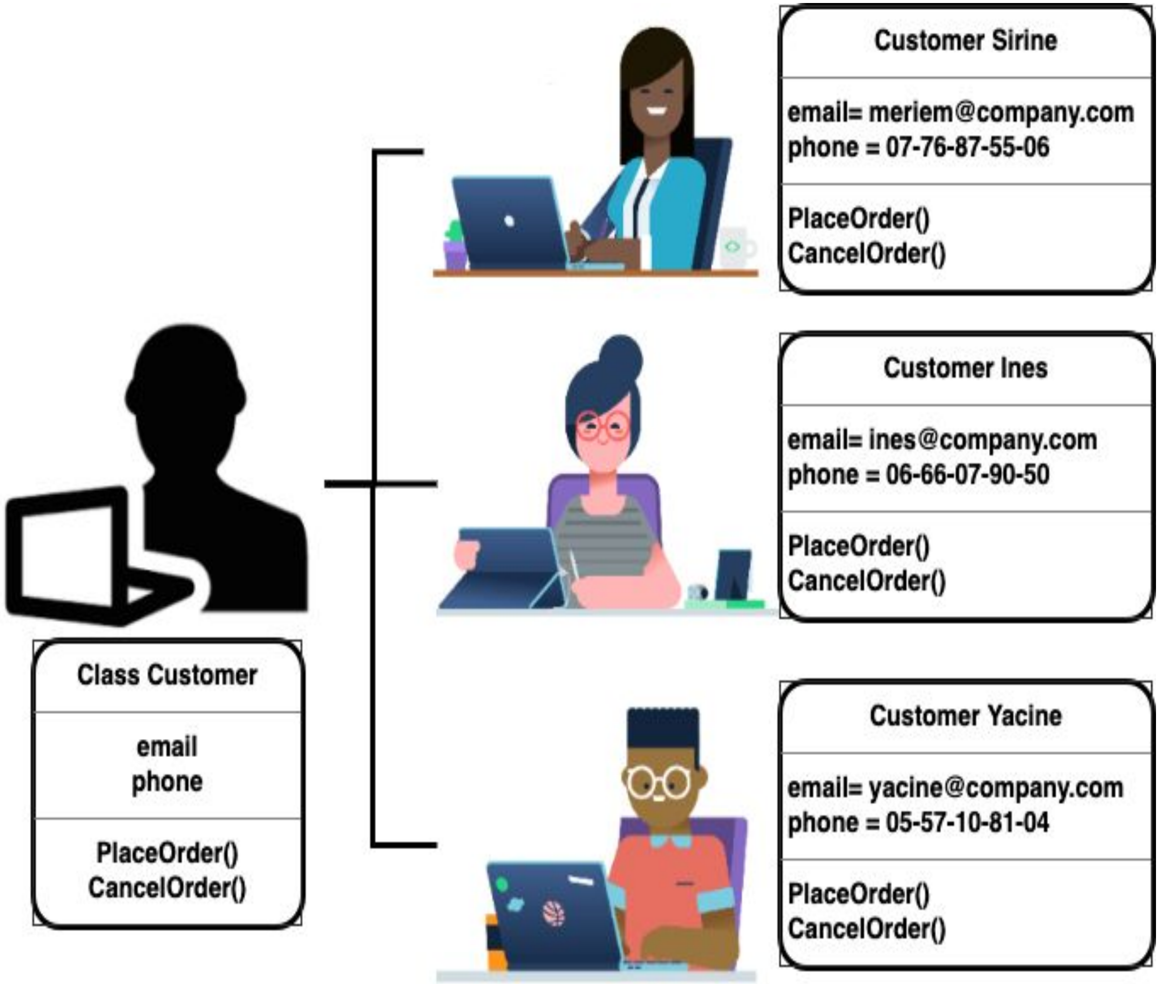
```python
import torch

class TinyModel(torch.nn.Module):

    def __init__(self):
        super(TinyModel, self).__init__()

        self.linear1 = torch.nn.Linear(100, 200)
        self.activation = torch.nn.ReLU()
        self.linear2 = torch.nn.Linear(200, 10)
        self.softmax = torch.nn.Softmax()

    def forward(self, x):
        x = self.linear1(x)
        x = self.activation(x)
        x = self.linear2(x)
        x = self.softmax(x)
        return x

tinymodel = TinyModel()
```

Modularity: Object-Oriented Programming

Fundamental Concepts

- The fundamental concepts of **OOP** are **classes** and **objects**.
- **Classes** ⇒ **blueprint** for objects describing possible **states** and **behaviors** bundled together.
- **Object** ⇒ a just a **specific realization** (**instantiation**) of a **class** with particular **state values**.

We think of **actions** and **data** as one unit representing a class ⇒ **Encapsulation**



Modularity: Object-Oriented Programming

Objects in Python

- Everything in **Python** is an **object** ⇒ **Numbers**, **Strings**, **DataFrames**, even **functions**
- Every **Object** has a **Class**
- Use **type()** to find the **class**

```
import numpy as np
a = np.array([1,2,3,4])
# print the type of a
print(type(a))
```

```
<class 'numpy.ndarray'>
```

Object	Class
5.2	float
“Welcome”	str
pd.DataFrame()	DataFrame
np.sum	function
[1,2,3]	list
…	…

Modularity: Object-Oriented Programming

Objects in Python: State & Behavior

- Object = State (attributes) + Behavior (methods)
  - State ⟺ variable ⟺ obj.my_attribute
  - Behavior ⟺ function() ⟺ obj.my_method()

State ⟺ Attributes

```
import numpy as np
a = np.array([1,2,3,4])
# shape attribute
a.shape
```

```
(4,)
```

All attributes and methods of an object

```
# list all attributes and methods
dir(a)
```

```
['T',
 '__abs__',
 '__add__',
 '__and__',
 '__array__',
```

Behavior ⟺ Methods

```
import numpy as np
a = np.array([1,2,3,4])
# reshape method
a.reshape(2,2)
```

```
array([[1, 2],
       [3, 4]])
```


Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Modularity: Object-Oriented Programming

Objects in Python: Basic Class

- **class <name>**: starts a class definition
- code inside class is **indented**
- **Attributes** are defined by assignment inside the **constructor**
- **method** definition ⇒ **function** definition within class
- use **self** to represent the instance of the class
  - used to access class' **attributes** and **methods**
  - should be the **1st argument** of any class method
  - Automatically handled by python when calling methods
- **Attributes** are created once the object is created.
- **Attributes** can be created in **methods**, but will be accessible only one the method is called **(not recommended)**

Constructor (Class Initiator)

Adding Attributes (Created once an object is instantiated)

Adding Methods (Behavior)

```
class MyClass:
    def __init__(self, attr1, attr2):
        self.attr1 = attr1
        self.attr2 = attr2
        # ...

    def my_method1(self, par1):
        self.attr1 = par1
        #....

    def my_method2(self, par2):
        self.attr2 = par2
        #...
```

```
# Instantiate obj from MyClass class
obj = MyClass(3, "Hi")
# Access attributes and methods
print(obj.attr2)
obj.my_method1(5)
print(obj.attr1)
```

```
Hi
5
```


Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Modularity: Object-Oriented Programming

Objects in Python: Core Principles

- **Instance-level data**
  - **name** , **salary** are **instance attributes**
- **Class-level data:**
  - Useful for **Global Constants** related to the class
  - **MIN_SALARY** is shared among all instances
  - Don't use **self** to define **class attributes**
- Printing just the value of an **Employee** object (**emp1**) ⇒ will print the reference to **@mem** allocated to the object.
  - We need a better **representative** meaning when printing an object

```python
class Employee:
    # Define a class attribute
    MIN_SALARY = 30000 #<--- no self.

    def __init__(self, name, salary):
        self.name = name
        # Use class name to access class attribute
        if salary >= Employee.MIN_SALARY:
            self.salary = salary
        else:
            self.salary = Employee.MIN_SALARY
```

```python
# Instantiating two Employee objects
emp1 = Employee("Ahmed SLIMANI", 40000)
emp2 = Employee("Mourad ABIDLI", 65000)
# printing
print(f"Hi, My name is {emp1.name} with min salary:{emp1.MIN_SALARY}")
print(f"Hi, My name is {emp2.name} with min salary:{emp2.MIN_SALARY}")
```

```
Hi, My name is Ahmed SLIMANI with min salary:30000
Hi, My name is Mourad ABIDLI with min salary:30000
```

```python
# print object
print(emp1)
```

```
<__main__.Employee object at 0x7fd55fce6210>
```

Modularity: Object-Oriented Programming

Objects in Python: Core Principles

- **Printing an object**
  - **__str__()**
    - print(obj) , str(obj)

```
import numpy as np
print(np.array([1,2,3]))
str(np.array([1,2,3]))
```

```
[1 2 3]
'[1 2 3]'
```

  - Informal, for **end user**
  - **String representation**

- **Printing an object**
  - **__repr__()**
    - repr(obj)

```
import numpy as np

repr(np.array([1,2,3]))
```

```
'array([1, 2, 3])'
```

  - Formal, for **developer**
  - Reproducible representation

Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Modularity: Object-Oriented Programming

Objects in Python: Core Principles

- **Implementing __str__()**

- Calling **print(object)** ⇒ Will implicitly call **__str__()**

```python
        else:
            self.salary = Employee.MIN_SALARY

    # reimplementing __str__ for printing
    def __str__(self):
        emp_str = """Employee:
          name: {name}
          salary: {salary}""".format(name=self.name,salary=self.salary)
        return emp_str
```

```python
emp = Employee("Ahmed SLIMANI", 40000)
print(emp)
```

```
Employee:
          name: Ahmed SLIMANI
          salary: 40000
```


Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Modularity: Object-Oriented Programming

Objects in Python: Core Principles

- **Implementing __repr__()**
  - Calling just the **object** ⇒ Will implicitly call **__repr__()**
  - Surround string arguments with quotation marks in the **__repr__()** to represent better the output

```python
class Employee:
    # Define a class attribute
    MIN_SALARY = 30000 #<--- no self.

    def __init__(self, name, salary):
        self.name = name
        # Use class name to access class attribute
        if salary >= Employee.MIN_SALARY:
            self.salary = salary
        else:
            self.salary = Employee.MIN_SALARY

    # reimplementing __repr__ for printing
    def __repr__(self):
        return """Employee('{name}', {salary})""".format(name=self.name,salary=self.salary)
```

```python
emp = Employee("Ahmed SLIMANI", 40000)
emp
```

```
Employee('Ahmed SLIMANI', 40000)
```


Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Modularity: Object-Oriented Programming

Objects in Python: Core Principles

- **Object Equality:**
  - emp1 and emp2 are not equal even they contain the same content
  - Why? ⇒ Here, equality is asserted based on @mem

```
emp1 = Employee("Ahmed SLIMANI", 40000)
emp2 = Employee("Ahmed SLIMANI", 40000)
# assert equality
emp1 == emp2
```

```
False
```

```
print(emp1)
print(emp2)
```

```
<__main__.Employee object at 0x7f33136453d0>
<__main__.Employee object at 0x7f3313645290>
```

- **Solution: Overloading __eq__()**
  - Implement **__eq__()** in your class ⇒ Called when 2 objects of a class are compared using ==.
  - Accepts 2 arguments, **self** and an **other** object to compare.

```
    # implementing __eq__()
    def __eq__(self, other):
        # Diagnostic printout
        print("__eq__() is called")
        # Returns True if all attributes match
        return (self.name == other.name) and \
               (self.salary == other.salary)
```

```
emp1 = Employee("Ahmed SLIMANI", 40000)
emp2 = Employee("Ahmed SLIMANI", 40000)
# assert equality
emp1 == emp2
```

```
__eq__() is called
True
```

Modularity: Object-Oriented Programming

Objects in Python: Core Principles

- **Comparison Operators**

Operator	Method
==	__eq__()
!=	__ne__()
>=	__ge__()
<=	__le__()
>	__gt__()
<	__lt__()

Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Modularity: Object-Oriented Programming

Objects in Python: Core Principles

- **Exception**
  - **Exceptions** are **classes**
  - Prevent the program from terminating when raised
  - Many **built-in Exceptions** in Python:

```
BaseException
+-- Exception
    +-- ArithmeticError
    |    +-- FloatingPointError
    |    +-- OverflowError
    |    +-- ZeroDivisionError
    +-- TypeError
    +-- ValueError
    |    +-- UnicodeError
    |         +-- UnicodeDecodeError
    |         +-- UnicodeEncodeError
    |         +-- UnicodeTranslateError
    +-- RuntimeError
   ...
+-- SystemExit
...
```

- **Exception Handling**

```python
try:
   # do something
   pass
except ValueError:
   # handle ValueError exception
   pass
except (TypeError, ZeroDivisionError):
   # handle multiple exceptions
   # TypeError and ZeroDivisionError
   pass
except:
   # handle all other exceptions
   pass
finally: # Optional
   # Run this code no matter what
   pass
```

Modularity: Object-Oriented Programming

Objects in Python: Core Principles

- **Exception Handling Example**
  - **Exceptions** are **classes**
  - Prevent the program from terminating when raised
  - Many **built-in Exceptions** in Python:

```
def  increase_salary_rate(self, cur_salary, salary_increase):
  try:
      return 100*(salary_increase / cur_salary)
  except (ZeroDivisionError, TypeError, ValueError) as e:
      print(e)
```

```
emp = Employee("Ahmed SLIMANI", 40000)
emp.increase_salary_rate(0, 10)
```

```
division by zero
```

- **Raising Exception**

```
def __init__(self, name, salary):
    try:
        if(name==''):
            raise Exception('name must be not empty')
        else:
            self.name = name
    except Exception as e:
        self.name = 'Anonymous'
        print(e)
    # Use   class name to access class attribute
    if salary >= Employee.MIN_SALARY:
        self.salary = salary
    else:
        self.salary = Employee.MIN_SALARY
```

```
emp = Employee("", 40000)
emp
```

```
name must be not empty
Employee('Anonymous', 40000)
```

Modularity: Object-Oriented Programming

Objects in Python: Core Principles

- **Class Inheritance & Polymorphism**
  - **Code reuse:**
    - **OOP** is fundamentally about **code reuse**
    - Millions of people out there writing code to solve parts of our problems
    - **Modules** are great for fixed functionally (code reuse)
  - What if that code doesn't **match** your needs exactly?
    - **OOP** is great for **customizing functionality ⇒ Inheritance**

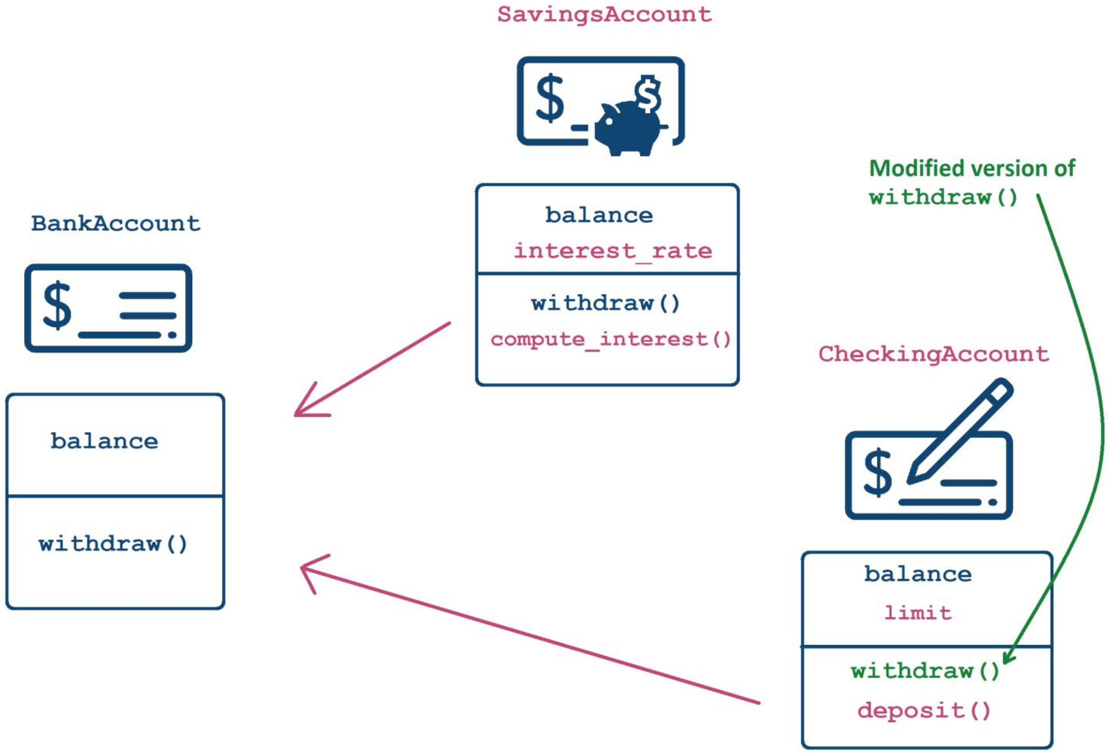
New class functionality = Old class functionality + extra

Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Modularity: Object-Oriented Programming

Objects in Python: Core Principles

- **Class Inheritance & Polymorphism**
  - **Example**
    - A basic **BankAccount** class that has a balance attribute and a withdraw method.
    - By **inheritance** ⇒ several types of accounts can be created via **code reuse**: **SavingAccount** and **CheckingAccount**:
      - Add new **attributes** or **methods**
      - Modify version of existed **methods**



Modularity: Object-Oriented Programming

Objects in Python: Core Principles

- **Class Inheritance & Polymorphism**
  - **Implementing Class Inheritance: Basics**

```
class BankAccount:
  def __init__(self, balance):
      self.balance = balance

  def withdraw(self, amount):
      self.balance -= amount

# Empty class inherited from BankAccount
class SavingsAccount(BankAccount):
    pass
```

- **Inheritance: "is-a" relationship**
  - A **SavingsAccount** is a **BankAccount** (Possibly with extra functionalities)

```
savings_acct = SavingsAccount(1000)
print(isinstance(savings_acct, SavingsAccount))
print(isinstance(savings_acct, BankAccount))
```

```
True
True
```

```
acct = BankAccount(500)
print(isinstance(acct,SavingsAccount))
print(isinstance(acct,BankAccount))
```

```
False
True
```


Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Modularity: Object-Oriented Programming

Objects in Python: Core Principles

- **Class Inheritance & Polymorphism**
    - **Customizing Constructors:**
        - Can run constructor of the parent class first by Parent **__init__(self, args...)**
        - Add more functionality as usual
        - Can use the data from both the **parent** and the **child** class

```
class SavingsAccount(BankAccount):
  """ Constructor speficially for SavingsAccount
      with an additional attribut"""
  def __init__(self, balance, interest_rate):
    # Call the parent constructor using ClassName.
    BankAccount.__init__(self, balance)
    # Add more functionality
    self.interest_rate = interest_rate
```

```
saving_acc = SavingsAccount(200, 0.5)
interest = saving_acc.compute_interest()
print(interest)
```

```
100.0
```


Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Modularity: Object-Oriented Programming

Objects in Python: Core Principles

- **Class Inheritance & Polymorphism**
  - **Customizing Constructors:**
    - Can run constructor of the parent class first by Parent **__init__(self, args...)**
    - Add more functionality as usual
    - Can use the data from both the **parent** and the **child** class
    - Can override existing methods ⇒ **Polymorphism**

```python
class CheckingAccount(BankAccount):
    def __init__(self, balance, limit):
        BankAccount.__init__(self, balance)
        # Add more functionality
        self.limit = limit
    # New method
    def deposit(self, amount):
        self.balance += amount
    # Modify method
    def withdraw(self, amount, fee=0):
        if fee <= self.limit:
            BankAccount.withdraw(self, amount - fee)
        else:
            BankAccount.withdraw(self,
                                 amount - self.limit)
```

```python
check_acct = CheckingAccount(1000, 25)
# Will call withdraw from CheckingAccount
check_acct.withdraw(200)
```


Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Modularity: Object-Oriented Programming

Objects in Python: Core Principles

- **Protected Access: @property**
  - Use "**protected**" attribute with leading _ to store data.
  - Use **@property** on a method whose name is exactly the name of the **restricted attribute**; return the internal attribute.
  - Use **@attr.setter** on a method **attr()** that will be called on **obj.attr = value**
- The value to assign passed as argument

```python
class Employee:
    def __init__(self, name, new_salary):
        self._salary = new_salary

    @property
    def salary(self):
        return self._salary

    @salary.setter
    def salary(self, new_salary):
        if new_salary < 0:
            raise ValueError("Invalid salary")
        self._salary = new_salary
```

```python
emp = Employee("Miriam Azari", 35000)
# accessing the "property"
emp.salary
```

```
35000
```

```python
emp.salary = 60000 # <-- @salary.setter
```

```python
emp.salary = -100
```

```
ValueError: Invalid salary
```

Modularity – Modules

What is a Python Module?

- A module ⇒ A logical organization of a Python Code so that similar piece of codes are grouped into a single file.

- A module ⇒ A single namespace, with a collection of functions, constants, class definitions and variables grouped in a single file.

- **How?**
  - Collect classes and functions in library modules
  - just put classes and functions in a file **module_name.py**
  - Put **module_name.py** in one of the directories where Python can find it.

employeeAccount.py ×

Users › macbook › Desktop › seds-labs › employeeAccount.py › ...

```python
# Employee class Creation
class Employee:⋯

# BankAccount class Creation
class BankAccount:
    def __init__(self, balance):
        self.balance = balance

    def withdraw(self, amount):
        self.balance -= amount

# SavingAccount class inherited from BankAccount
class SavingsAccount(BankAccount):⋯

# CheckingAccount class inherited from BankAccount
class CheckingAccount(BankAccount):⋯
```


Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Modularity – Modules

Importing Python Modules

- “**import**” statement ⇒ most common way to use modules in Python Code.
- To import entire module
  - **import** <module name>
  - Example: **import** math
- To import specific function/object from module:
  - **from** <module_name> **import** <function_name>
  - Example: **from** math **import** sqrt
- We may use **import aliasing** for abbreviation purposes using the word **as**.

Note: Python comes with hundreds of built-in modules. math is one among them. You may refer to https://docs.python.org/3/py-modindex.html for the complete python list modules.

```python
# import module
import employeeAccount
# call modules classes and Methods
saving_acc = employeeAccount.SavingsAccount(200, 0.5)
interest = saving_acc.compute_interest()
print(interest)
```

```python
# import module using aliasing
import employeeAccount as ea
# use the alias to call classes and Methods
saving_acc = ea.SavingsAccount(200, 0.5)
interest = saving_acc.compute_interest()
print(interest)
```

```python
# import a class from a module
from employeeAccount import SavingsAccount
# call the classe and its Methods
saving_acc = SavingsAccount(200, 0.5)
interest = saving_acc.compute_interest()
print(interest)
```

Modularity – Modules

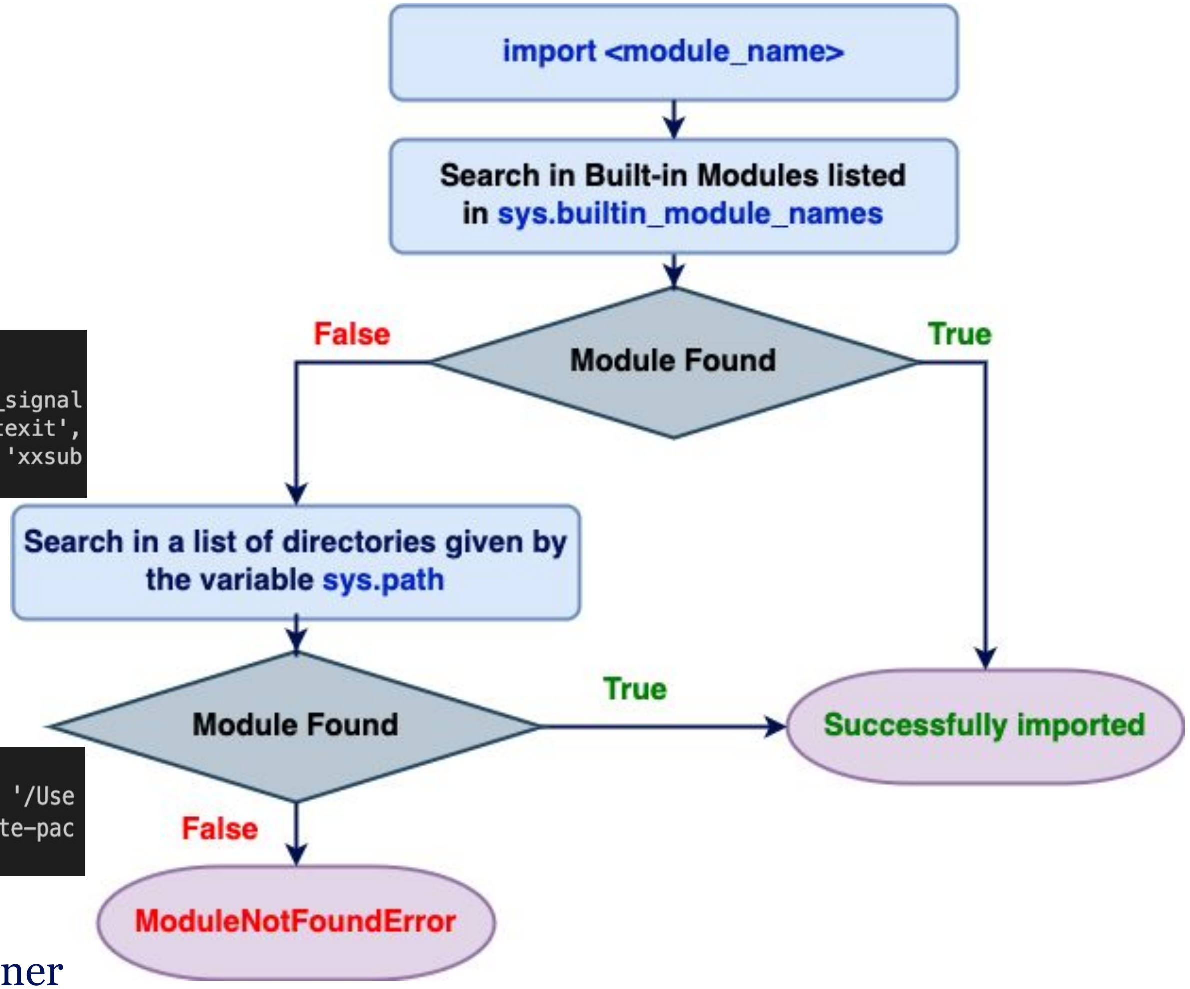
More on Modules

- **The Module Search Path:**
  - When a module is imported:
    - The interpreter first searches for a built-in module with that name.

```
>>> import sys
>>> sys.builtin_module_names
('_abc', '_ast', '_codecs', '_collections', '_functools', '_imp', '_io', '_locale', '_operator', '_signal
', '_sre', '_stat', '_string', '_symtable', '_thread', '_tracemalloc', '_warnings', '_weakref', 'atexit',
 'builtins', 'errno', 'faulthandler', 'gc', 'itertools', 'marshal', 'posix', 'pwd', 'sys', 'time', 'xxsub
type')
```

    - If not found, it then searches in a list of directories given by the variable sys.path. sys.

```
>>> sys.path
['', '/Users/macbook/opt/anaconda3/lib/python38.zip', '/Users/macbook/opt/anaconda3/lib/python3.8', '/Use
rs/macbook/opt/anaconda3/lib/python3.8/lib-dynload', '/Users/macbook/opt/anaconda3/lib/python3.8/site-pac
kages', '/Users/macbook/opt/anaconda3/lib/python3.8/site-packages/aeosa']
```



Note: A **__pycache__** directory will be created in the container directory, if one does not already exist, to speed up loading modules.

Modularity – Modules

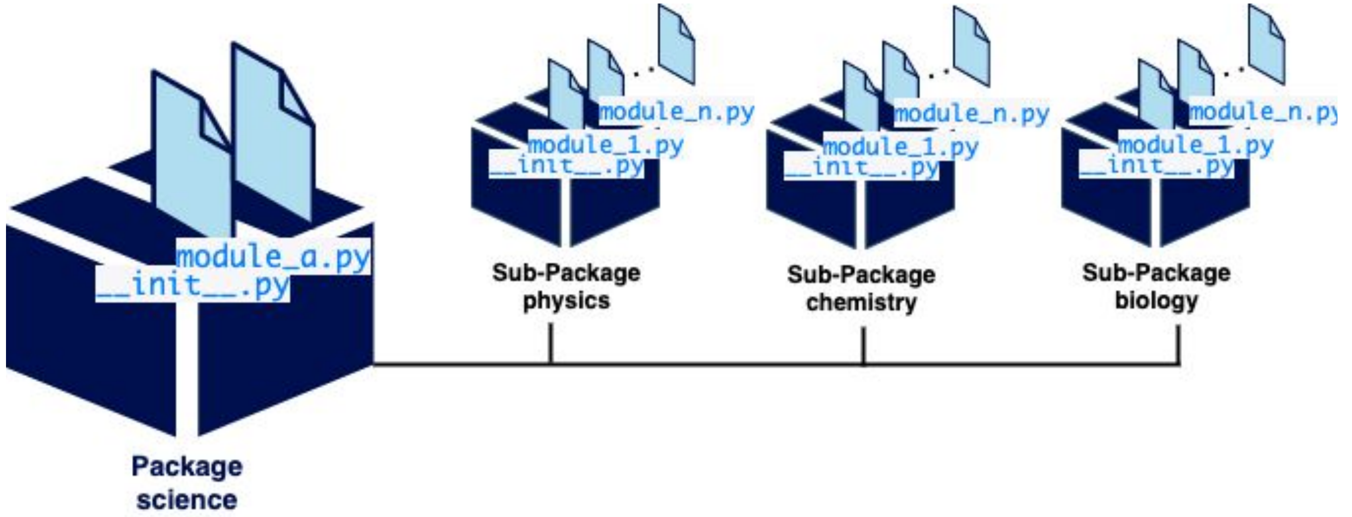
More on Modules

- **Ensuring your Module is Found:**
  - **Using Current Directory:**
    - Put **<module_name.py>** in the directory where the input script is located (the current directory)
  - **Using PYTHONPATH:**
    - Modify the **PYTHONPATH** environment variable to contain the directory where **<module_name.py>** is located before starting the interpreter.
    - Or put **<module_name.py>** in one of the directories already contained in the **PYTHONPATH** variable.
  - **Using the installation-dependent directories:**
    - Put **<module_name.py>**in one of the installation-dependent directories, which you may or may not have write-access to, depending on the OS.
  - **Using any Directory:**
    - Put **<module_name.py>** in any directory of your choice and then modify **sys.path** at run-time

Modularity – Packages

What is a Python package?

- A “**package**” ⇒ a module, except it can have other modules (and indeed other packages) inside it.
- A **package** ⇒ a directory with a **__init__.py** file and any number of python files or other package directories:
- The **__init__.py** can be:
  - Totally empty – or
  - Have arbitrary python code in it.
    - The code will be run when the package is imported.
      - Idem while importing a nested package.
- **Not:** Modules or sub-packages inside packages are not automatically imported.



- **import science**
  - will run the code in **science/__init__.py**.
- Any names defined in the **__init__.py** will be available in: **science.a_name**But,
- **science.module_a**
  - will not exist. To get submodules, you need to explicitly import them
  - **import science.module_a**

Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

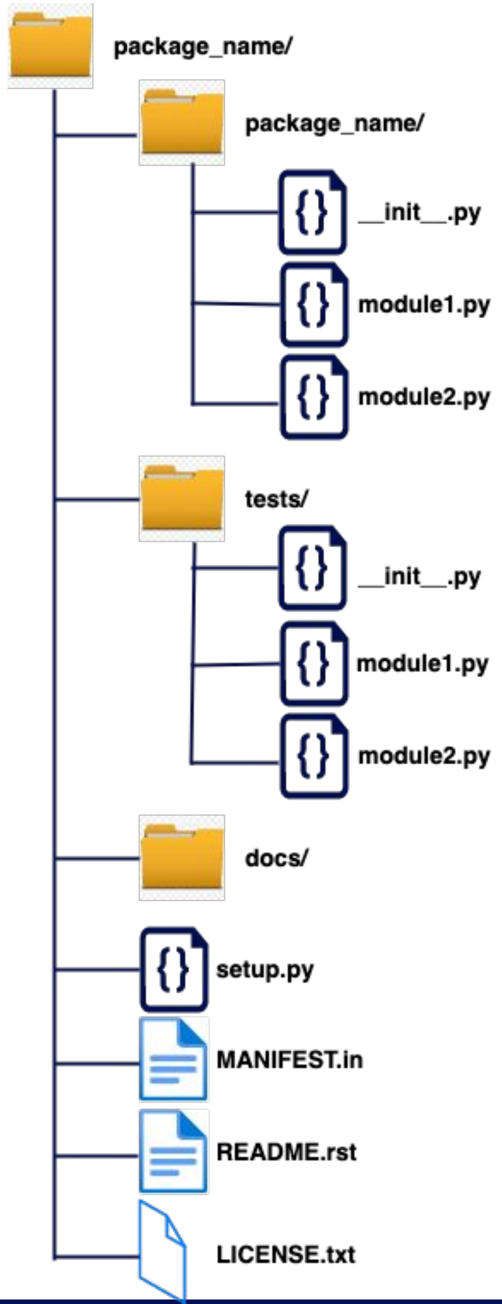
Modularity – Packages

Building Your Own Package

- Use a well structured, standard layout for your package to help you build, install and distribute it.
- Have to be standardized for publishing in the Python Package Index (**PyPI**) repository.
- Create the structure
  - Manually or
  - Use specific python packages
    - **Cookiecutter**

Basic Package Structure:

package_name/	The main package – this is where the code goes
tests/	your unit tests (Recommended).
docs/	documentation (Recommended).
LICENSE.txt	text of the license you choose (do choose one!) (Optional).
MANIFEST.in	description of what non-code files to include (Optional).
README.rst (.txt, .md)	description of the package – should be written in ReST or Markdown (for **PyPi**) (Recommended).
setup.py	the script for building/installing package (Mandatory).



Packages

Building Your Own Package: The LICENSE.txt File

- **LICENSE.txt** file ⇒ A text file licensing your package for anybody to:

Use **Modify** **Share**

For any purpose

Subject to conditions preserving the provenance and openness of the package

- Most well-known open source licences:
  - GNU AGPLv3
  - MIT

Note: More details on open source Licenses can be found in https://choosealicense.com/licenses/

Example: Copy and Past the MIT License Content available in https://choosealicense.com/licenses/mit/ to create a LICENSE.txt.

LICENSE.txt ×

Users > macbook > Desktop > employeeaccount > LICENSE.txt

```
MIT License

Copyright (c) [year] [fullname]

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

Modularity – Packages

Building Your Own Package: The setup.py File

- **setup.py** file ⇒ A declarative Python file describing your package in terms of:
  - Version & package metadata
  - List of packages to include
  - List of other files to include
  - List of dependencies
  - List of extensions to be compiled
- It tells **setuptools** how to package, build and install it.
  - **Setuptools** ⇒ A package development process library designed for creating and distributing Python packages.

setup.py

Users > macbook > Desktop > seds-labs > setup.py > ...

```python
from setuptools import setup

setup(
    name='PackageName',
    version='0.1.0',
    author='An Awesome Coder',
    author_email='author_email@esi-sba.com',
    packages=['package_name', 'package_name.test'],
    url='http://pypi.python.org/pypi/PackageName/',
    license='LICENSE.txt',
    description='An awesome package that does something',
    long_description=open('README.txt').read(),
    install_requires=[
        "pandas == 0.20.3",
        "pytest",
    ],
)
```

Modularity – Packages

Building Your Own Package: The README.txt File

- **README.txt** file ⇒ A descriptive manual file for your Python package project, alternatively named **README.rst** or **README.md**.
- To be displayed properly on **PyPI**, choose a markup language supported by PyPI:
  - **plain text**
  - **reStructuredText**
  - **Markdown**

- A good **README** file should include:
  - A descriptive project title
  - Motivation (why the project exists)
  - How to setup
  - Copy-pastable quick start code example
  - Recommended citation

Modularity – Packages

Building Your Own Package: The MANIFEST.in File (Optional)

- A built source distribution (**sdist**) ⇒ by default contains only a minimal set of files (All Python source files, …).
- We want to include extra files in the source distribution, such as non-code files ⇒ Solution use the **MANIFEST.in** file.
- **MANIFEST.in** file ⇒ A descriptive file for adding & removing files to & from the source distribution.
- A **MANIFEST.in** file ⇒ A set of commands, executed one per line, instructing **setuptools** to add or remove some set of files from the sdist.

Some MANIFEST.in Commandes

Command	Description
include pat1 pat2 …	Add all files matching any of the listed patterns
exclude pat1 pat2 …	Remove all files matching any of the listed patterns
global-include pat1 pat2 …	Add all files anywhere in the source tree matching any of the listed patterns
global-exclude pat1 pat2 …	Remove all files anywhere in the source tree matching any of the listed patterns
graft dir-pattern	Add all files under directories matching dir-pattern
prune dir-pattern	Remove all files under directories matching dir-pattern

Example:

```
graft tests
global-exclude *.py
```

→

1. Add the contents of the directory tree **tests** to the **sdist**
2. Remove all files in the sdist with a **.py** extension

Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Modularity – Packages

Building Your Own Package: The README.md File Example

Markdown Input

````
# package_name

`package_name` is a Python library for ..........

## Installation

Use the package manager [pip](https://pip.pypa.io/en/stable/) to install
`package_name`.

```bash
pip install package_name
```

## Usage

```python
import package_name

# returns 'words'
package_name.module_name('word')

# returns 'geese'
package_name.module_name('goose')

# returns 'phenomenon'
package_name.module_name('phenomena')
```

## License
[MIT](https://choosealicense.com/licenses/mit/)
````

Rendered Output

package_name

`package_name` is a Python library for ....

Installation

Use the package manager pip to install `package_name`.

```
pip install package_name
```

Usage

```
import package_name

# returns 'words'
package_name.module_name('word')

# returns 'geese'
package_name.module_name('goose')

# returns 'phenomenon'
package_name.module_name('phenomena')
```

License

MIT

Modularity – Packages

Building Your Own Package: Builds a Distribution

- With a **setup.py** script defined, **setuptools** can do a lot:
  - Build a source distribution (**sdist**) ⇒ a tar archive of all the files needed to build and install the package.
  - Builds wheels (**bdist_wheel**) ⇒ a binary distribution **.whl** file directly installable through the pip install command.
    - The same file to be uploaded to **pypi.org**
    - to upload your packgage to **pypi.org** you have to first register an account: https://pypi.org/account/register/.

Offline

```
python setup.py sdist
pip install dist/<file-tar-name.tar.gz>
```

Offline

```
python setup.py bdist_wheel
pip install dist/<file-wheel-name.whl>
```

Online

```
twine upload dist/*
pip install <package-name>
```

Note: The generated **sdist** or **.whl** files will be under subdirectory **dist/** of your package.

Twine package (have to be installed first) ⇒ provides a secure, authenticated, and verified connection between your system and PyPi over HTTPS.

Readability

Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Readability

Introducing PEP 8

- Pythonistas software engineering community has **conventions** of coding in Python ⇒ **Protocol 8 (PEP 8)**
- **PEP 8** is the defacto Style Guide for Python Code
  - Guide you how to format your code to be as readable as possible.
- **Example of Violating PEP 8**
  - The module **import** isn't at the top of the file
  - The **spacing** and **indentation** is inconsistent
  - The lack of **line breaks** makes it difficult to tell when one idea finishes and the next begins

```
#define our data
my_dict ={
'a' : 10,
'b': 3,
'c' : 4,
'd': 7}
#import needed package
import numpy as np
#helper function
def DictToArray(d):
    """Convert dictionary values to numpy array"""
    #extract values and convert
    x=np.array(d.values())
    return x
print(DictToArray(my_dict))
```


Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Readability

Introducing PEP 8

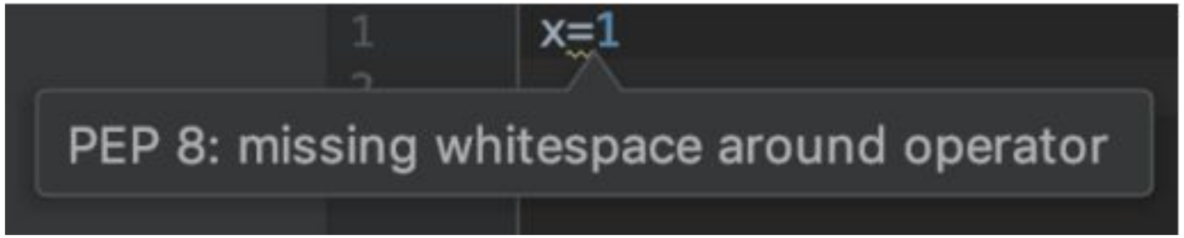
- **Example of Following PEP 8**
    - The same chunk of code looks much better after rewritten to conform to **PEP 8 conventions**.
        - ⇒ Following the agreed-upon rules in **PEP 8** such as using **spacing**, **indentation**, **break lines**, and others appropriately.
    - The code became much more readable despite accomplishing the same exact task.

```python
# Import needed package
import numpy as np

# Define our data
my_dict = {'a': 10,
           'b': 3,
           'c': 4,
           'd': 7}

# Helper function
def dict_to_array(d):
    """Convert dictionary values to numpy array"""
    # Extract values and convert
    x = np.array(d.values())
    return x


print(dict_to_array(my_dict))
```

Readability

Introducing PEP 8

- **PEP 8 Tools**
  - Many rules defined in **PEP 8** ⇒ We need tools that can check our code.
  - Smart IDEs can flag violations as soon as you write a bad line of code,
  - Other options ⇒ Use the **pycodestyle** package.
    - **Pycodestyle**
      - Check code in multiple files at once
      - Output descriptions of the violations along with the required information to fix the issue.



Readability

Introducing PEP 8

- **Using pycodestyle**
    - Installation:

```
pip install pycodestyle
```

```
conda install pycodestyle
```

    - Using pycodestyle:

```
pycodestyle <your_python_file.py>
```

File | Column # | Error Description

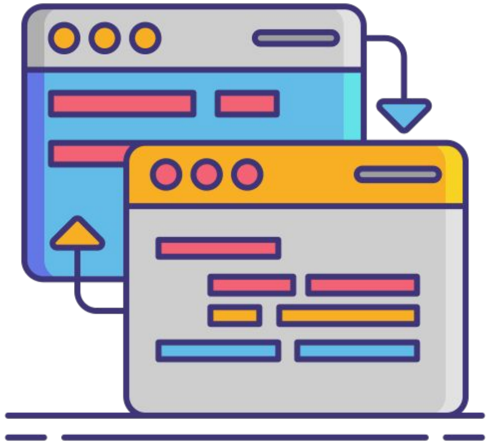
```
testpycodestyle.py:2:10: E225 missing whitespace around operator
```

Line # | Error Code

Example of using the code in **slide 36** when saved in a **testpycodestyle.py** file

```
testpycodestyle.py:1:1: E265 block comment should start with '# '
testpycodestyle.py:2:10: E225 missing whitespace around operator
testpycodestyle.py:3:1: E122 continuation line missing indentation or outdented
testpycodestyle.py:3:4: E203 whitespace before ':'
testpycodestyle.py:4:1: E122 continuation line missing indentation or outdented
testpycodestyle.py:5:1: E122 continuation line missing indentation or outdented
testpycodestyle.py:5:4: E203 whitespace before ':'
```


Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz



Refactoring

Refactoring

Refactoring Code

- **Refactoring** ⇒ Restructuring your code to improve its internal structure, without changing its external functionality.
  - The more you **refactor** your code ⇒ the best becomes **cleaner** and **modular**.
    - **Clean** ⇒ Readable, Simple, and Concise.
- **Why Refactoring?**
  - Provide a better built, well-structured, more readable code
  - Speed up your development time in the long run
  - Easier to maintain code
  - Reuse more of your code
  - Become a much better programmer

- **When Refactoring?**
  - Duplicated code
  - Long Method
  - Large Classes
  - Long Parameter List
  - Divergent Change
  - .....

Refactoring

Basic Tips for Writing Clean Code

- **Use Meaningful Names**
  - Be descriptive and consistent
  - Avoid abbreviations and especially single letters
- **Use Pre-built python functions packages**
  - Be efficient in coding

```python
# Student Test Scores
s = [16.5, 12.25, 10.5, 9.0]

# Printing The Average Student Test Scores
print(sum(s)/len(s))
```

⬇

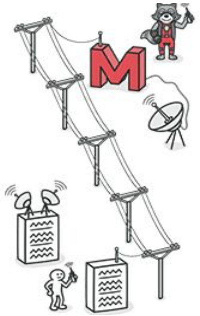
```python
import numpy as np
# Student Test Scores
test_scores = [16.5, 12.25, 10.5, 9.0]

# Printing The Average Student Test Scores
print(np.average(test_scores))
```

Refactoring

Useful Refactoring Techniques

Composing Methods

- Streamline methods, remove code duplication, and pave the way for future improvements.



Simplifying Method Calls

- Make method calls simpler and easier to understand.

Simplifying Conditional Expressions

- Conditionals tend to get more and more complicated in their logic over time, and there are yet more techniques to combat this as well.

Refactoring

Refactoring Techniques – Composing Methods: Examples

- **Extract Method**
  - **Problem** – You have a code fragment that can be grouped together.

```python
def printOwing(self):
    self.printBanner()

    # print details
    print("name:", self.name)
    print("amount:", self.getOutstanding())
```

  - **Solution** – Move this code to a separate new method (or function) and replace the old code with a call to the method.

```python
def printOwing(self):
    self.printBanner()
    self.printDetails(self.getOutstanding())

def printDetails(self, outstanding):
    print("name:", self.name)
    print("amount:", outstanding)
```

  - **Why?**
    - The more lines found in a method, the harder it's to figure out what the method does.

Refactoring

Refactoring Techniques – Composing Methods: Examples

- **Inline Method**
  - **Problem** – When a method body is more obvious than the method itself, use this technique.
  - **Solution** – Replace calls to the method with the method's content and delete the method itself.
  - **Why?**
    - A method simply delegates to another method. In itself, this delegation is no problem. But it may become a confusing in some cases.

```python
class PizzaDelivery:
    # ...
    def getRating(self):
        return 2 if self.moreThanFiveLateDeliveries() else 1

    def moreThanFiveLateDeliveries(self):
        return self.numberOfLateDeliveries > 5
```

```python
class PizzaDelivery:
  # ...
  def getRating(self):
    return 2 if self.numberOfLateDeliveries > 5 else 1
```

Refactoring

Refactoring Techniques – Composing Methods: Examples

- **Extract Variable**
  - **Problem** – You have an expression that's hard to understand.
  - **Solution** – Place the result of the expression or its parts in separate variables that are self-explanatory.
  - **Why?**
    - Make a complex expression more understandable, by dividing it into its intermediate parts.

```
def renderBanner(self):
    if (self.platform.toUpperCase().indexOf("MAC") > -1) and \
       (self.browser.toUpperCase().indexOf("IE") > -1) and \
       self.wasInitialized() and (self.resize > 0):
        # do something
        pass
```

```
def renderBanner(self):
    isMacOs = self.platform.toUpperCase().indexOf("MAC") > -1
    isIE = self.browser.toUpperCase().indexOf("IE") > -1
    wasResized = self.resize > 0

    if isMacOs and isIE and self.wasInitialized() and wasResized:
        # do something
        pass
```


Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Refactoring

Refactoring Techniques – Composing Methods: Examples

- **Inline Temp**
  - **Problem** – You have a temporary variable that's assigned the result of a simple expression and nothing more.

```
def hasDiscount(order):
    basePrice = order.basePrice()
    return basePrice > 1000
```

```
def hasDiscount(order):
    return order.basePrice() > 1000
```

  - **Solution** – Replace the references to the variable with the expression itself.
  - **Why?**
    - Marginally improve the readability of the code by getting rid of the unnecessary variable.

Refactoring

Refactoring Techniques – Composing Methods: Examples

- **Replace Temp with Query**
  - **Problem** – You place the result of an expression in a local variable for later use in your code.
  - **Solution** – Move the entire expression to a separate method and return the result from it. Query the method instead of using a variable. Incorporate the new method in other methods, if necessary.
  - **Why?**
    - The same expression may sometimes be found in other methods as well, which is one reason to consider creating a common method.

```
def calculateTotal():
    basePrice = quantity * itemPrice
    if basePrice > 1000:
        return basePrice * 0.95
    else:
        return basePrice * 0.98
```

```
def calculateTotal():
    if basePrice() > 1000:
        return basePrice() * 0.95
    else:
        return basePrice() * 0.98

def basePrice():
    return quantity * itemPrice
```


Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Refactoring

Refactoring Techniques – Composing Methods: Examples

- **Substitute Algorithm**
  - **Problem** – So you want to replace an existing algorithm with a new one?.
  - **Solution** – Replace the body of the method that implements the algorithm with a new algorithm.
  - **Why?**
    - To make sure that you have simplified the existing algorithm as much as possible.

```python
def foundPerson(people):
    for i in range(len(people)):
        if people[i] == "Don":
            return "Don"
        if people[i] == "John":
            return "John"
        if people[i] == "Kent":
            return "Kent"
    return ""
```

```python
def foundPerson(people):
    candidates = ["Don", "John", "Kent"]
    return people if people in candidates else ""
```


Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Refactoring

Refactoring Techniques – Simplifying Conditional Expressions: Examples

- **Consolidate Duplicate Conditional Fragments**
    - **Problem** – Identical code can be found in all branches of a conditional.

```
if isSpecialDeal():
    total = price * 0.95
    send()
else:
    total = price * 0.98
    send()
```

    - **Solution** – Move the code outside of the conditional.

```
if isSpecialDeal():
    total = price * 0.95
else:
    total = price * 0.98
send()
```

    - **Why?**
        - Code deduplication.

Refactoring

Refactoring Techniques – Simplifying Conditional Expressions: Examples

- **Consolidate Conditional Expression**
  - **Problem** – You have multiple conditionals that lead to the same result or action.
  - **Solution** – Consolidate all these conditionals in a single expression.
  - **Why?**
    - To eliminate duplicate control flow code and hence for greater clarity.

```
def disabilityAmount():
    if seniority < 2:
        return 0
    if monthsDisabled > 12:
        return 0
    if isPartTime:
        return 0
    # Compute the disability amount.
    # ...
```

```
def disabilityAmount():
    if isNotEligibleForDisability():
        return 0
    # Compute the disability amount.
    # ...
```

Refactoring

Refactoring Techniques – Simplifying Conditional Expressions: Examples

- **Replace Nested Conditional with Guard Clauses**
  - **Problem** – You have a group of nested conditionals and it's hard to determine the normal flow of code execution.
  - **Solution** – Isolate all special checks and edge cases into separate clauses and place them before the main checks.
  - **Why?**
    - To make it easy to figure out what each conditional does.

```python
def getPayAmount(self):
    if self.isDead:
        result = deadAmount()
    else:
        if self.isSeparated:
            result = separatedAmount()
        else:
            if self.isRetired:
                result = retiredAmount()
            else:
                result = normalPayAmount()
    return result
```

```python
def getPayAmount(self):
    if self.isDead:
        return deadAmount()
    if self.isSeparated:
        return separatedAmount()
    if self.isRetired:
        return retiredAmount()
    return normalPayAmount()
```

Refactoring

Refactoring Techniques – Simplifying Method Calls: Examples

- **Replace Parameter with Method Call**
  - **Problem** – Calling a query method and passing its results as the parameters of another method, while that method could call the query directly.
  - **Solution** – Instead of passing the value through a parameter, try placing a query call inside the method body.
  - **Why?**
    - To get rid of unneeded parameters and simplify method calls.

```
basePrice = quantity * itemPrice
seasonalDiscount = self.getSeasonalDiscount()
fees = self.getFees()
finalPrice = discountedPrice(basePrice,
                             seasonalDiscount,
                             fees)
```

```
basePrice = quantity * itemPrice
finalPrice = discountedPrice(basePrice)
```

Refactoring

Refactoring Techniques – Simplifying Method Calls: Examples

- **Replace Parameter with Explicit Methods**
    - **Problem** – A method is split into parts, each of which is run depending on the value of a parameter.
    - **Solution** – Extract the individual parts of the method into their own methods and call them instead of the original method.
    - **Why?**
        - To Improve code readability and much easier to understand the purpose of the method.

```
def setValue(self, name, value):
  if (name == "height"):
      self.height = value
      return

  if (name == "width"):
      self.width = value
      return
```

```
def setHeight(self, arg):
  self. height = arg

def setWidth(self, arg):
  self.width = arg
```


Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Refactoring

Refactoring Techniques – Simplifying Method Calls: Examples

- **Replace Error Code with Exception**
  - **Problem** – A method returns a special value that indicates an error?
  - **Solution** – Throw an exception instead.
  - **Why?**
    - Returning error codes is an obsolete holdover from procedural programming.
    - To frees cod from a large number of conditionals for checking various error codes.

```
def withdraw(self, amount):
    if amount > self.balance:
        return -1
    else:
        self.balance -= amount
    return 0
```

```
def withdraw(self, amount):
    if amount > self.balance:
        raise BalanceException()
    self.balance -= amount
```

Thanks for your Listening

Dr. Belkacem KHALDI
e-mail: b.khaldi@esi-sba.dz

Refactoring

Tips for Writing Clean Code

- **Use Meaningful Names**
  - Be descriptive and imply type
  - Be consistent but clearly differentiate
  - Avoid abbreviations and especially single letters
- **Use Pre-built python functions packages**
  - Be efficient in coding

```
# Student Test Scores Coef.
c = [4, 3, 2, 2]

# Student Weighted Test Scores (test Scores multiplied by Coef.).
w = []
for i,j in zip(s,c):
   w.append(i*j)

# Printing The Average Student Test Scores
print(sum(w)/sum(c))
```

```
# Student Test Scores Coef.
test_scores_coefs = [4, 3, 2, 2]

# Student Weighted Test Scores (test Scores multiplied by Coef.).
weighted_test_scores = [i * j for i, j in zip(test_scores,test_scores_coefs)]

# Printing The Average Student Test Scores
print(sum(weighted_test_scores)/sum(test_scores_coefs))
```

```
# Printing The Average Student Test Scores
print(np.average(test_scores, weights = test_scores_coefs))
```

Refactoring

Tips for Writing Clean Code

- **Use Modular Code**
  - Avoide code repetition
  - Reuse your code

```python
import numpy as np
# Student Test Scores
test_scores = [16.5, 12.25, 10.5, 9.0]

# Student Test Scores Coef.
test_scores_coefs = [4, 3, 2, 2]

# Printing The Average Student Test Scores
if (test_scores_coefs == None):
    print(np.average(test_scores))
else:
    print(np.average(test_scores, weights = test_scores_coefs))
```

⬇

```python
import numpy as np
# Student Test Scores
test_scores = [16.5, 12.25, 10.5, 9.0]

# Student Test Scores Coef.
test_scores_coefs = [4, 3, 2, 2]
# Printing The Average Student Test Scores
print(np.average(test_scores) if test_scores_coefs == None else np.average(test_scores, weights = test_scores_coefs))
```

Refactoring

Tips for Writing Clean Code

- **Use Modular Code**
  - Don't Repeat Yourself

```python
import numpy as np
# Student Test Scores
test_scores= [16.5, 12.25, 10.5, 9.0]

# Adding 0.5 mark to all scores
test_scores_1= [x + 0.5 for x in test_scores]

# Adding 2 mark to all scores
test_scores_2= [x + 2 for x in test_scores]

# Multiplying all scores by 1.125
test_scores_3= [x * 1.125 for x in test_scores]

# Printing the results
print(np.mean(test_scores_1))
print(np.mean(test_scores_2))
print(np.mean(test_scores_3))
```

  - Functions should do one thing

```python
import numpy as np
# Student Test Scores
test_scores= [16.5, 12.25, 10.5, 9.0]

# A function to add a given mark to all scores
def score_adding(_scores, _a = 0.5):
    return [score + _a for score in _scores]

# A function to multiply all scores by a given rate
def score_multiplying(_scores, _m = 1.125):
    return [score * _m for score in _scores]

# Printing the results
for avg_score in [np.mean(score_adding(test_scores)),
                  np.mean(score_adding(test_scores, 2)),
                  np.mean(score_multiplying(test_scores))]:
    print(avg_score)
```