Python Advanced

Agenda

Inheritance

Inheritance basics

1. Classes can inherit fields and methods from other classes
2. The class that inherits after another is called a subclass
3. The class that is inherited by another class is called a superclass or a baseclass
4. If a field/method of baseclass is not re-defined in subclass, it becomes defined
5. If a field/method of baseclass is re-defined in subclass, it is overridden.
6. Subclass can call methods from its parent using super() object
7. The syntax is **class** SubclassName(SuperclassName):

Example 01-inheritance/example-01.py

```python
class Animal:
        def __init__(self, name):
                self._name = name
        @property
        def name(self):
                return f"My name is {self._name}"

class Dog(Animal):
        def __init__(self, name):
                super().__init__(name)
        @property
        def name(self):
                return f"My name is {self._name}, the dog."


if __name__ == "__main__":
        fido = Dog("Fido")
        print(fido.name)
```

```
$ python3 01-inheritance/example-01.py

  My name is Fido, the dog.
```

Inheritance introspection

1. A subclass is an instance of all its superclasses
2. We can check if an instance's class/subclass using isinstance() built-in function
3. We can check whether also check if class is a subclass of another using issubclass()function

Example 01-inheritance/example-02.py

```python
class Animal:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return f"My name is {self._name}"

class Dog(Animal):
    def __init__(self, name):
        super().__init__(name)


    @property
    def name(self):
        return f"My name is {self._name}, the dog."


if __name__ == "__main__":
        print(f"Dog is a subclass of Animal: {issubclass(Dog, Animal)}")
        print(f"Animal is a subclass of object: {issubclass(Animal, object)}")
        d = Dog("Rex")
        print(f"d is an instance of Dog: {isinstance(d, Dog)}")
        print(f"d is an instance of Animal: {isinstance(d, Animal)}")
        print(f"d is an instance of object: {isinstance(d, object)}")
```

```
$ python3 01-inheritance/example-02.py

Dog is a subclass of Animal: True
Animal is a subclass of object: True
d is an instance of Dog: True
d is an instance of Animal: True
d is an instance of object: True
```

Inheritance order

1. As seen in the previous example, subclasses can become baseclasses for their own subclasses
2. Since everything in Python is an object, everything is a subclass/instance of something else
3. All classes in Python inherit from object by default: class A: is equivalent to **class** A(object):
4. To initialize all the superclasses properly, we need to call their `__init__()` functions, by calling `super().__init__()`

Example 01-inheritance/example-03.py

```python
class A:
    def __init__(self):
        print(f"A init called")

class B(A):
    def __init__(self):
        super().__init__()
        print(f"B init called")


class C(B):
    def __init__(self):
        print(f"C init called")
        super().__init__()


class D(A):
    def __init__(self):
        super().__init__()
        print(f"D init called")


if __name__ == "__main__":
        print("Creating an instance of class C)
        c = C()
        print("Creating an instance of class D)
        d = D()
```

```
$ python3 01-inheritance/example-03.py


Creating an instance of class C
C init called
A init called
B init called
Creating an instance of class D
A init called
D init called
```

Exercise

Try calling super first in C.__init__, what happens?   Try doing the same in D.__init__

Exercise 2

Using Animal and Dog as an example:

1. Add make_a_sound() method to both of them, Animal should return an empty string, while a Dog should return "woof!"
2. Write another class - Cat and implement make_a_sound() for it too
3. Add a new field to Animal class called age and update Animal's init to take it as an argument.
4. Update calls to super in both subclasses to include age

Multiple inheritance

1. In Python, classes can inherit after more than one class
2. When they do that, they take on properties of all their superclasses
3. This method is often used to extend class’s abilities using smaller, functionality-oriented classes.

Example 01-inheritance/example-04.py

```python
import json
class JSONOutput:
    def __init__(self, *args, **kwargs):
            pass

    def to_json(self):
            return json.dumps(self)

class SimpleRow(dict, JSONOutput):
    def __init__(self, *args, **kwargs):
            super().__init__(*args, **kwargs)

    def _headers(self):
            header_width = max(len(str(k)) for k in self)
            headers = " | ".join(str(k).center(header_width) for k in self)
            return f"| {headers} |"
    def _values(self):
            header_width = max(len(str(k)) for k in self)
            values = " | ".join(str(v).center(header_width) for v in self.values())
            return f"| {values} |"
    def table(self):
            return f"{self._headers()}\n{self._values()}"
if __name__ == "__main__":
            row = SimpleRow(no=1, name="Mark", surname="O'Connor", nationality="Irish")
            print(row.table())
            print(row.to_json())
```

```
$ python3 01-inheritance/example-04.py

| no | name | surname  | nationality |
| 1  | Mark | O'Connor |    Irish    |

{"no": 1, "name": "Mark", "surname":
"O'Connor", "nationality": "Irish"}
```

Exercise 3

1. Using *example-04.py* write a class called **SimpleTable** which is a subclass of **list** and **JSONOutput** (in that order)
2. Since **SimpleTable** inherits from list, you can append objects to it.
3. Append several **SimpleRow** objects to it.
4. **SimpleTable** should implement a function called table() which output is similar to **SimpleRow**.table(), but instead it displays first row's headers and values and only values for the other rows
5. Test whether to_json() works for SimpleTable

Method Resolution Order*

This is an advanced topic.

1. What happens when Python inherits from classes that inherit from the same class?
2. This situation is sometimes called *diamond inheritance*
3. Python provides a MRO - method resolution order
4. MRO is an algorithm which decides the order of called superclasses
5. MRO is also used when calling a function to decide which of the re-defined functions will be called
6. You can inspect class's MRO by either Class.__mro__ or Class.mro()
7. Method resolution order can get quite complex
8. Further reading https://www.python.org/download/releases/2.3/mro/

Example 01-inheritance/example-05.py

```python
class A:
    def __init__(self):
        print("A.__init__")
    def func1(self):
        print("A.func1()")
    def func2(self):
        print("A.func2()")
    def func3(self):
        print("A.func3()")

class B(A):
    def __init__(self):
        super().__init__()
        print("B.__init__")
    def func1(self): print("B.func1()")

class C(A):
    def __init__(self):
        super().__init__()
        print("C.__init__")
    def func2(self):
        print("C.func2()")
```

```python
class D(B, C):
    def __init__(self):
        super().__init__()
        print("D.__init__")

    def func3(self): p
        print("D.func3()")

    def super_func3(self):
        super().func3()

if __name__ == "__main__":
    d = D()
    d.func1()
    d.func2()
    d.func3()
    d.super_func3()
```

```
$ python3 01-inheritance/example-05.py

A.__init__
C.__init__
B.__init__
D.__init__
B.func1()
C.func2()
D.func3()
A.func3()
```

Operator overloading

Inheritance recap

Inheritance recap:

1. classes can inherit after one another
2. classes can override each other’s methods

Knowing that, we can override special methods used by Python operators to achieve our own goals

Example 02-overloading/example-01.py

```python
class EshopCart:
    def __init__(self, buyer):
        self.buyer = buyer
        self.products = []
        self.total = 0.0

    def add_product(self, name, price):
        self.products.append(name)
        self.total += price

    def __len__(self):
        return len(self.products)

if __name__ == "__main__":
    cart = EshopCart("Ann")
    cart.add_product("jeans", 30.0)
    print(f"Cart's length: {len(cart)}")
```

```
$ python3 02-overloading/example-01.py

  Cart's length: 1
```

Exercise 1

1. Override addition operator:
   - It should return the left-side object at the end
   - It should add right-side object's products and total to left-side's fields

2. Override membership operator:
   - It should return True if given product can be found in cart's products list

3. Override greater-than, less-than and equal operators:
   - They should compare self.total values

4. Override __str__ to return a pretty result: EshopCart{buyer: X, total: Y, products: Z}
   - X is buyer's name
   - Y is cart's total value
   - Z is number of products in cart

Class Instantiation part II

__new__ vs __init__

1. Before calling __**init**__ on the newly created class instance, Python calls __**new**__
2. __**new**__ is object's class method used to create the new instance in the first place
3. Although it is not recommended to override __**new**__, we will do so to understand how it works
4. To display the mechanisms in new, we will implement a Singleton Pattern

Singelton (anti)pattern

1. Singleton is a name of *design pattern*
2. Singleton pattern is quite controversial in itself, but it is a good showcase of overriding **__new__**
3. Singleton is defined as such:
    1. Singleton is a class, which can posses only one instance
    2. If an instance of the class does not exist yet, it creates a new instance
    3. If an instance exists, it returns the existing instance instead

Example 03-class-instantiation/example-01.py

```python
class Singleton:
    _instance = None

    def __init__(self, name):
        print(f"- Instance initialization: name={name}")
        self.name = name

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            print("- Creating new instance")
            cls._instance = object.__new__(cls)
        print("- Returning existing instance")
        return cls._instance

if __name__ == "__main__":
    print("A ".ljust(30, "-"))
    s = Singleton("A")
    print("B ".ljust(30, "-"))
    s2 = Singleton("B")
    print("C ".ljust(30, "-"))
    s3 = Singleton("C")
    print(f"They are all the same object: {s is s2 is s3}")
    print(f"Singleton._instance.name: {Singleton._instance.name}")
```

```
$ python3 03-class-instantiation/example-01.py

 A -----------------------------
 - Creating new instance
 - Returning existing instance
 - Instance initialization: name=A
 B -----------------------------
 -  Returning existing instance
 -  Instance initialization: name=B
 C -----------------------------
 - Returning existing instance
 - Instance initialization: name=C

 They are all the same object: True

 Singleton._instance.name: C
```

Decorators

Decorators

1. Decorator is a design pattern used to add functionality to an existing object without modifying the object.
2. In Python you can define decorators using:
    - a function
    - a class

Function-based decorators

1. A function-based decorator is a function which:
   - takes the function that needs to be decorated as an argument
   - returns another function, which uses the decorated function inside it
2. Decorating function f with decorator d is equivalent to d(f)
3. It is common for the returned function to be decorated with **functools.wraps**, so that the name of decorated function and its docstring are preserved.

Example 04-decorators/example-01.py

```python
import functools

    def example_decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            print("Wrapper: Before function execution")
            result = func(*args, **kwargs)
            print("Wrapper: After function execution")
            return result

    return wrapper

@example_decorator
def greetings(name):
    print(f"Hello, {name}!")

if __name__ == "__main__":
    greetings("Jane")
```

```
$ python3 04-decorators/example-01.py

 Wrapper: Before function execution
 Hello, Jane!
 Wrapper: After function execution
```

Exercise

Given function **random_string** which returns a string:

1. Write a decorator which lowercases any string returned by a decorated function
2. Write a decorator which shortens a string to 40 characters if it is longer than that
3. Decorate random_string with both and call it a few times.

Class-based decorators

A class-based decorator is a class which:

1. takes the function that needs to be decorated as an argument
2. implements __**call**__ special method to become a callable object
3. calls stored function inside __**call**__ implementation

Example 04-decorators/example-02.py

```python
class HTMLHeader:

    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        result = self.func(*args, **kwargs)
        return f"<h1>{result}</h1>"

@HTMLHeader
    def prepare_title(title_string):
        return title_string.title()

if __name__ == "__main__":
    print(prepare_title("this is a section title!"))
```

```
$ python3 04-decorators/example-02.py

 <h1>This Is A Section Title!</h1>
```

Exercise 2

Given function power(x, y), which calculates x to the power of y 1 million times:

1. Write a class-based decorator which measures time it took to execute wrapped
2. function and prints it
3. Hint: use time() function from time library
4. Call power() with large numbers to see the results (15.0 and 60.0 will do fine)

Parametric decorators

1. Parametric decorators are decorators which have arguments themselves
2. They can be either class- or function-based

Example 04-decorators/example-03.py

```python
import functools

def html_tag(tag_name):

    def wrapper(func):

        @functools.wraps(func)
        def wrap(*args, **kwargs):
            result = func(*args, **kwargs)
            return f"<{tag_name}>{result}</{tag_name}>"

        return wrap

    return wrapper @html_tag("h1")

def prepare_h1_title(title_string):
    return title_string.title()

@html_tag("h2")
def prepare_h2_title(title_string):
    return title_string.lower()

if __name__ == "__main__":
    print(prepare_h1_title("a H1 title!"))
    print(prepare_h2_title("a H2 subtitle!"))
```

```
$ python3 04-decorators/example-03.py

 <h1>A H1 Title!</h1>
 <h2>a h2 subtitle!</h2>
```

Context managers

Context managers

1. Python supports a special type of decorator called *context manager*
2. You have been using one of them, you were just unaware that it's called that
3. Context managers can take parameters
4. Context managers usually execute stuff both on entry and on exit
5. Context managers can be called using **with** ... syntax
6. You can define both function- and class-based context managers
7. __**enter**__ is called upon entering **with** ... block
8. __**exit**__ is called upon leaving **with** ... block

Example 05-context-managers/example-01.py

```python
class FileOpen:
    """FileOpen is an illustration of Context Manager’s
    body, it does almost exactly what
    `with open() as ...` does"""

    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
        self.opened_file = None

    def __enter__(self):
        print("Opening file")
        self.opened_file = open(self.filename, self.mode)
        return self.opened_file # this enables us to use `as ...`

    def __exit__(self, *exc):
        print("Safely closing file") self.opened_file.close()

if __name__ == "__main__":
    path = "05-context-managers/example-01.py"
    with FileOpen(path, "r") as f:
    print(f"{path} has {len(f.readlines())} lines")
    print("Finished")
```

```
$ python3 05-context-managers/example-01.py

Opening file
05-context-managers/example-01.py has 25 lines
Safely closing file
Finished
```

Example 05-context-managers/example-02.py

```python
"""example-02.py reimplements FileOpen from example-01.py as a function-based context manager"""
import contextlib

@contextlib.contextmanager
def file_open(filename, mode):
    print("Opening file")
    f = open(filename, mode)
    print("Starting with... block")
    yield f # yield keyword will be discussed more closely in 'Generators' module
    print("Closing file")
    f.close()

if __name__ == "__main__":
    path = "05-context-managers/example-01.py"
    with file_open(path, "r") as f:
    print(f"{path} has {len(f.readlines())} lines")
    print("Finished")
```

```
$ python3 05-context-managers/example-01.py

  Opening file
  Starting with... block
  05-context-managers/example-01.py has 25 lines
  Closing file
  Finished
```

Exercise 1

Write a **timeit** context manager which saves a current **time.time()** value on entry and prints time passed since entry on exit. Try splitting the following chunk of text a million times inside the context to test it. Use punctuation mark as a delimiter.

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore
eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt
in culpa qui officia deserunt mollit anim id est laborum.
```

Lambda functions

Lambda functions

1. So far all the functions we defined had a name - they were named functions
2. Python also supports *anonymous* functions
3. Anonymous functions in Python are called *lambdas*
4. Lambda’s syntax can be described **lambda** <**arguments**>**:** <**operation**>
5. Lambda returns the result of operation by default

Example 06-lambdas/example-01.py

```python
fruit = ["apples", "bananas", "oranges"]
capitalized_fruit = list(map(lambda s: s.capitalize(), fruit))
print(fruit)
print(capitalized_fruit)
```

```
$ python3 06-lambdas/example-01.py

['apples', 'bananas', 'oranges’]
['Apples', 'Bananas', 'Oranges']
```

Lambda functions

Lambdas are commonly used in places where defining a function would be too much. Since lambdas are *anonymous* functions, they should never be assigned to any identifier.

Exercise 1

1. Given a list of User objects, try sorting them using:
   - registered_on
   - number_of_logins
   - last_seen
2. To do that, use either list.sort() or sorted(). Both support a keyword parameter **key=callable**
3. Result of callable will be compared using comparison operators to determine order in list
4. Use **lambda** as **callable**

Map function

1. Map function is used to apply a function to every element of an iterable, as shown in **example-01.py**
2. It returns a generator
3. Syntax: **map(function, iterable)**

Exercise 2

1. Given list **digits = [1, 2, 5, 9],** create a list of their squares using map() and a lambda.
2. Given list **fruit = ["apples", "oranges", "grapes"]**, create a dictionary, where the string is the key, and the value is string’s length. Use map() and a **lambda**.

Filter function

1. filter function is used to filter out iterable elements, for which a given function returns **False**
2. It returns a generator
3. Syntax: **filter(function, iterable)**

Example 06-lambdas/example-02.py

```python
even_numbers = filter(lambda num: num % 2 == 0, range(1, 25))
print(list(even_numbers))
```

```
$ python3 06-lambdas/example-02.py
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24]
```

Reduce function

1. reduce is part of the functools library
2. reduce applies given two-argument function to its current state and first unused element of iterable
3. If its state is not initialized using initial parameter, it uses two first elements of iterable instead of initial state and first element.
4. For example reduce(lambda x, y: x+y, [1, 2, 3]) could be described as ((1+2) + 3)
5. reduce(lambda x, y: x+y, [1, 2, 3], 5) could be described as (((5 + 1) + 2) + 3)
6. Syntax: reduce(function, iterable, initial=None)

Example 06-lambdas/example-03.py

```python
import functools

numbers = [1, 2, 3, 4, 5, 6]
sum_of_numbers = functools.reduce(lambda x, y: x + y, numbers)
print(f"Sum of {numbers} equals {sum_of_numbers}")

words = ["This", "is", "a", "list", "of", "words"]
print(functools.reduce(lambda x, y: f"{x} {y}", words))

minimum_of_numbers = functools.reduce(min, numbers, -3)
print(f"Found minimum is {minimum_of_numbers}")
```

```
$ python3 06-lambdas/example-03.py

Sum of [1, 2, 3, 4, 5, 6] equals 21
This is a list of words
Found minimum is -3
```

Map-filter-reduce

1. Map-filter-reduce is a pattern commonly used in functional programming
2. Map is used to extract fields from a class instance or a dictionary
3. Filter is used to filter the results as desired
4. Reduce is used to aggregate values into a result

Example 06-lambdas/example-04.py

```python
from functools import reduce

class Car:
    def __init__(self, make, model, price):
        self.make = make
        self.model = model
        self.price = price

cars = [
            Car("Ford", "Anglia", 300.0),
            Car("Ford", "Cortina", 700.0),
            Car("Alfa Romeo", "Stradale 33", 190.0),
            Car("Alfa Romeo", "Giulia", 500.0),
            Car("Citroën", "2CV", 75.0),
            Car("Citroën", "Dyane", 105.0)
]
# Let's find the price of the cheapest Citroën on the list
c = reduce(min, map(lambda car: car.price, filter(lambda car: car.make == "Citroën", cars)) )
print(f"The cheapest Citroën costs: {c}")
```

```
$ python3 06-lambdas/example-04.py

The cheapest Citroën costs: 75.0
```

Exercise 3

1. Given car list from *example-04.py*, find out the following using reduce-filter or map- filter-reduce:
2. What is the total cost of all Alfa Romeos on the list?
3. What is the count of all Fords on the list? Hint: use `initial = 0`

Logging

Logging

1. While print function is very useful, it should not be used for reporting events that happen during script's exectution
2. This is because print function's output is buffered, which means it is not written to output immediately.
3. In such cases, one should use logging library and the Loggers it provides.
4. Each message can be logged on different logging level:
   1. DEBUG - detailed information regarding implementation details
   2. INFO - confirmation of application working as expected
   3. WARNING - indication of a possible problem, which however did not affect application's execution
   4. ERROR - indication that an action could not be performed due to a serious problem
   5. CRITICAL - usually indicates that the application cannot continue running

Example 07-logging/example-01.py

```python
import logging

if __name__ == "__main__":
    logging.debug(f"{dir(logging)[:10]}")
    logging.info("An information")
    logging.warning("A warning")
    logging.error("An error")
    logging.critical("Something is very wrong!")
```

```
$ python3 07-logging/example-01.py

WARNING:root:A warning
ERROR:root:An error
CRITICAL:root:Something is very wrong!
```

Logging levels

1. As you can see, we did not get all the messages logged in *example-01*
2. This is because we were using logger’s default configuration
3. By default, logging is set to warning level, which means logs less important than a warning will be suppressed.
4. We can configure logger’s level, output (console, file, both, multiple files), format and so on
5. Logging also enables you to create separate loggers with separate settings
6. To do that, use **logging.getLoggger(name)**.
7. Whenever you supply a name, you will get the logger instance attached to that name
8. If name is not specified - root logger is returned

Example 07-logging/example-02.py

```python
import logging
import tempfile

if __name__ == "__main__":
    l = logging.getLogger()
    l.setLevel(logging.DEBUG)
    console_handler = logging.StreamHandler()
    console_handler.setFormatter(
        logging.Formatter("%(levelname)-8s (%(asctime)s):%(message)s")
    )
    l.addHandler(console_handler)

    file_handler = logging.FileHandler("example-02.log", mode="w+")
    file_handler.setLevel(logging.ERROR)
    file_handler.setFormatter(
        logging.Formatter("%(levelname)-8s in %(filename)s: %(message)s")
    )
    l.addHandler(file_handler)

    logging.debug(f"{dir(logging)[:10]}")
    logging.info("An information")
    logging.warning("A warning")
    logging.error("An error")
    logging.critical("Something is very wrong!")

    print("\nexample-02.log contents:".ljust(50, "-"))
    with open("example-02.log") as f:
        for l in f:
            print(l, end="")
```

```
$ python3 07-logging/example-02.py

 example-02.log contents:--------------------
 ERROR    in example-02.py: An error
 CRITICAL in example-02.py: Something is very
 wrong!

 DEBUG    (2023-01-23 19:27:20,820):
 ['BASIC_FORMAT', 'BufferingFormatter',
 'CRITICAL', 'DEBUG', 'ERROR', 'FATAL',
 'FileHandler', 'Filter', 'Filterer',
 'Formatter']
 INFO     (2023-01-23 19:27:20,820): An
 information
 WARNING  (2023-01-23 19:27:20,820): A warning
 ERROR    (2023-01-23 19:27:20,820): An error
 CRITICAL (2023-01-23 19:27:20,820): Something
 is very wrong!
```

Exercise 1

Given file exercise-01.py

1. Convert all calls to print() to proper logging
2. configure logging to log simultaneously to two files and console, using the following formats and levels:
    - File A: timestamp - level - message level DEBUG
    - File B: filename - funcname - line number level INFO
    - Console: asctime [level]: message level WARN
3. refer to https://docs.python.org/3/library/logging.html#logrecord-attributes for attribute names

Generators

Iterators

1. Iterators are an object, which can be iterated upon. For example by a for loop
2. Iterator must implement two methods **__iter__** and **__next__**
3. Iterator returns its items one by one
4. When there are no more objects, it raises **StopIteration** error
5. Collections which can be return iterator are called *iterables*
6. Most of built-in collections are iterables