Python Intermediate

© 2019 Software Development Academy All Rights Reserved

Agenda

1. Inheritance
2. Operator overloading
3. Class instantiation part II
4. Decorators
5. Context managers
6. Lambda functions
7. Logging
8. Generators
9. Exceptions
10. Threading
11. Multiprocessing
12. Common serialization formats
13. Regular expressions

© 2019 Software Development Academy All Rights Reserved

Inheritance

© 2019 Software Development Academy All Rights Reserved

Inheritance basics

1. Classes can **inherit** fields and methods after 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):`

© 2019 Software Development Academy All Rights Reserved

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.
```

© 2019 Software Development Academy All Rights Reserved

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

© 2019 Software Development Academy All Rights Reserved

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
```

© 2019 Software Development Academy All Rights Reserved

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__()`

© 2019 Software Development Academy All Rights Reserved

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
```

© 2019 Software Development Academy All Rights Reserved

Exercise 1

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

© 2019 Software Development Academy All Rights Reserved

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`

© 2019 Software Development Academy All Rights Reserved

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.

© 2019 Software Development Academy All Rights Reserved

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"}
```

© 2019 Software Development Academy All Rights Reserved

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`

© 2019 Software Development Academy All Rights Reserved

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

© 2019 Software Development Academy All Rights Reserved

Example 01-inheritance/example-04.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):
        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
```

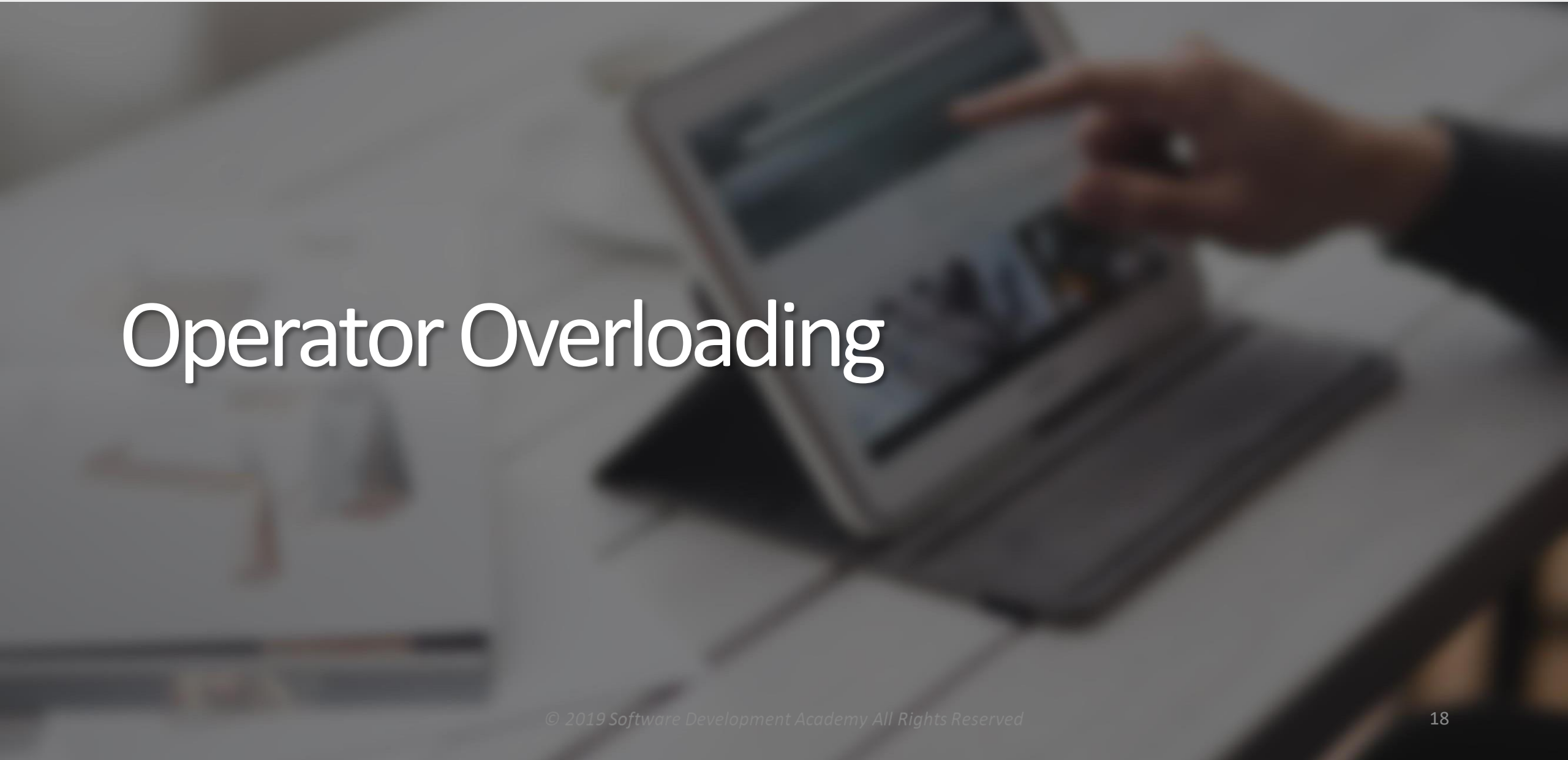
```
A.__init__
C.__init__
B.__init__
D.__init__
B.func1()
C.func2()
D.func3()
A.func3()
```

© 2019 Software Development Academy All Rights Reserved

Method Resolution Order*

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/

© 2019 Software Development Academy All Rights Reserved



Operator Overloading

© 2019 Software Development Academy All Rights Reserved

Inheritance recap

1. Inheritance recap:
   - classes can inherit after one another
   - classes can override each other's methods
2. Knowing that, we can override special methods used by Python operators to achieve our own goals

© 2019 Software Development Academy All Rights Reserved

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
```

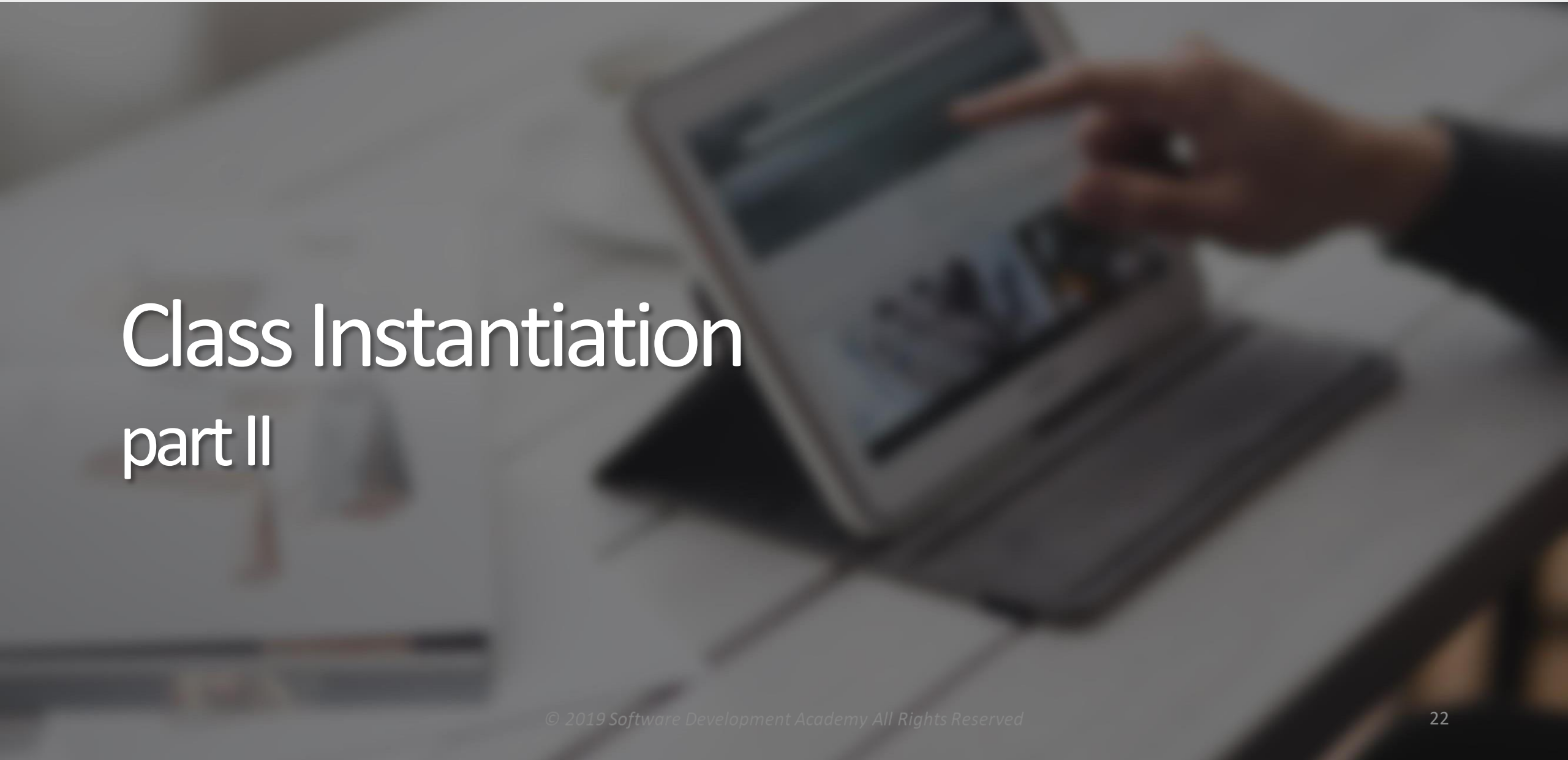
```
Cart's length: 1
```

© 2019 Software Development Academy All Rights Reserved

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 the cart

© 2019 Software Development Academy All Rights Reserved



Class Instantiation

part II

© 2019 Software Development Academy All Rights Reserved

__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

© 2019 Software Development Academy All Rights Reserved

Singleton (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

© 2019 Software Development Academy All Rights Reserved

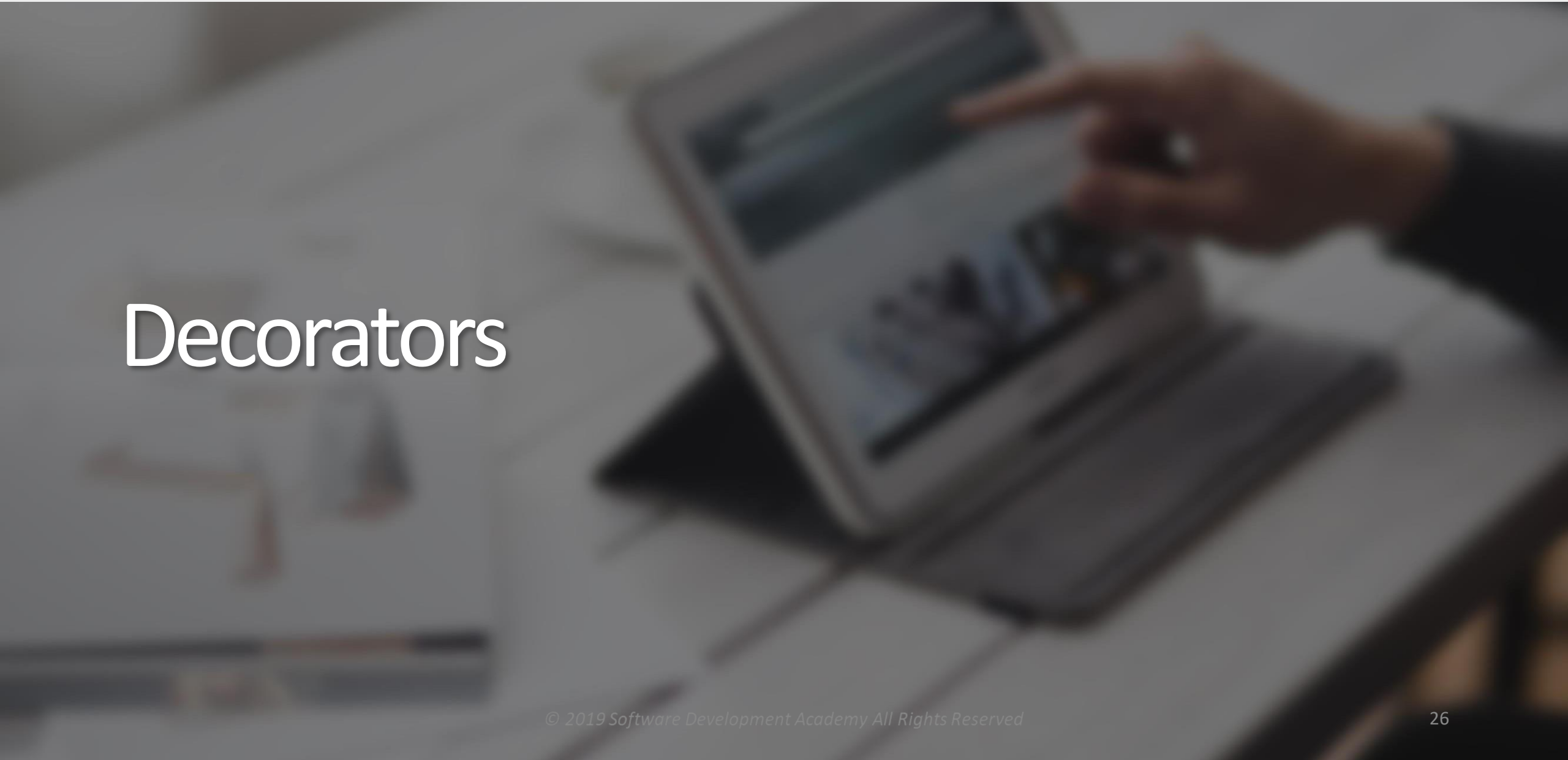
Example 03-class-instantiation/example-01.py

```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
```

© 2019 Software Development Academy All Rights Reserved



Decorators

© 2019 Software Development Academy All Rights Reserved

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

© 2019 Software Development Academy All Rights Reserved

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.

© 2019 Software Development Academy All Rights Reserved

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
```

© 2019 Software Development Academy All Rights Reserved

Exercise 1

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.

© 2019 Software Development Academy All Rights Reserved

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

© 2019 Software Development Academy All Rights Reserved

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>
```

© 2019 Software Development Academy All Rights Reserved

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 function and prints it
2. Hint: use `time()` function from `time` library
3. Call `power()` with large numbers to see the results (`15.0` and `60.0` will do fine)

© 2019 Software Development Academy All Rights Reserved

Parametric decorators

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

© 2019 Software Development Academy All Rights Reserved

Example 04-decorators/example-03.py

```
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>
```

© 2019 Software Development Academy All Rights Reserved

Context Managers

© 2019 Software Development Academy All Rights Reserved

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`

© 2019 Software Development Academy All Rights Reserved

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
```

© 2019 Software Development Academy All Rights Reserved

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

```
"""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-02.py
```

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

© 2019 Software Development Academy All Rights Reserved

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.
```

© 2019 Software Development Academy All Rights Reserved

Lambda functions

© 2019 Software Development Academy All Rights Reserved

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

© 2019 Software Development Academy All Rights Reserved

Example 06-lambdas/example-01.py

```
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']
```

© 2019 Software Development Academy All Rights Reserved

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.

© 2019 Software Development Academy All Rights Reserved

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`

© 2019 Software Development Academy All Rights Reserved

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)`

© 2019 Software Development Academy All Rights Reserved

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`.

© 2019 Software Development Academy All Rights Reserved

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)`

© 2019 Software Development Academy All Rights Reserved

Example 06-lambdas/example-02.py

```
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]
```

© 2019 Software Development Academy All Rights Reserved

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)`

© 2019 Software Development Academy All Rights Reserved

Example 06-lambdas/example-03.py

```
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
```

© 2019 Software Development Academy All Rights Reserved

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

© 2019 Software Development Academy All Rights Reserved

Example 06-lambdas/example-04.py

```
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
```

© 2019 Software Development Academy All Rights Reserved

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`

© 2019 Software Development Academy All Rights Reserved

Logging

© 2019 Software Development Academy All Rights Reserved

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:
    - `DEBUG` - detailed information regarding implementation details
    - `INFO` - confirmation of application working as expected
    - `WARNING` - indication of a possible problem, which however did not affect application's execution
    - `ERROR` - indication that an action could not be performed due to a serious problem
    - `CRITICAL` - usually indicates that the application cannot continue running

© 2019 Software Development Academy All Rights Reserved

Example 07-logging/example-01.py

```
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!
```

© 2019 Software Development Academy All Rights Reserved

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

© 2019 Software Development Academy All Rights Reserved

Example 07-logging/example-02.py

```
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
```

```
DEBUG    (2019-09-13 17:38:44,365): ['BASIC_FORMAT',
'BufferingFormatter', 'CRITICAL', 'DEBUG', 'ERROR',
'FATAL', 'FileHandler', 'Filter', 'Filterer', 'Formatter']
INFO     (2019-09-13 17:38:44,365): An information
WARNING  (2019-09-13 17:38:44,365): A warning
ERROR    (2019-09-13 17:38:44,365): An error
CRITICAL (2019-09-13 17:38:44,365): Something is very
wrong!

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

© 2019 Software Development Academy All Rights Reserved

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

© 2019 Software Development Academy All Rights Reserved

Generators

© 2019 Software Development Academy All Rights Reserved

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

© 2019 Software Development Academy All Rights Reserved

Example 08-generators/example-01.py

```
fruit = ["apple", "banana", "orange"]
fruit_iterator = iter(fruit)  # since lists are iterables, this returns an iterator
print(fruit_iterator)

print(next(fruit_iterator))
print(next(fruit_iterator))
print(fruit_iterator.__next__())
print(next(fruit_iterator))
```

```
$ python3 08-generators/example-01.py
```

```
<list_iterator object at 0x0000000002ce0cf8>
apple
banana
orange
Traceback (most recent call last):
  File "08-generators/example-01.py", line 8, in <module>
    print(next(fruit_iterator))
StopIteration
```

© 2019 Software Development Academy All Rights Reserved

Iterators - looping

1. For loops are a more elegant way of looping through an iterable
2. This way you don't need to handle `StopIteration` error

© 2019 Software Development Academy All Rights Reserved

Example 08-generators/example-02.py

```
fruit = ["apple", "banana", "orange"]
fruit_iterator = iter(fruit)  # since lists are iterables, this returns an iterator
for f in fruit_iterator:
    print(f)
```

```
$ python3 08-generators/example-02.py
```

```
apple
banana
orange
```

© 2019 Software Development Academy All Rights Reserved

Example 08-generators/example-03.py

```python
# Implementation of a custom iterator
class ZeroToN:
    def __init__(self, n):
        self.n = n

    def __iter__(self):  # called when iterator is created
        self.i = -1
        return self

    def __next__(self):  # called on next
        self.i += 1
        if self.i < self.n:
            return self.i
        else:
            raise StopIteration


for i in ZeroToN(10):
    print(i)
```

```
$ python3 08-generators/example-03.py
```

```
0
1
(…)
```

© 2019 Software Development Academy All Rights Reserved

Exercise 1

Using *example-03.py* as reference, implement iterator `MyRange` which behaves just like built-in `range(start, stop, step)`

© 2019 Software Development Academy All Rights Reserved

Generators

1. Generators are a simple way of creating iterators in Python
2. A generator executes until it reaches a `yield` statement, which halts its execution and may return a value
3. By calling `next()` on a iterator, which is a result of the generator, generator's execution is resumed until another `yield` statement is met
4. If there are no more `yield` statements, a `StopIteration` exception is raised by the iterator
5. Generators have many different uses, besides iterator generation: you have last seen them in the module about context managers, they are also widely used in asynchronous operations.

© 2019 Software Development Academy All Rights Reserved

Example 08-generators/example-04.py

```python
def my_range(start, stop, step=1):
    position = start
    while position < stop:
        yield position
        position += step


def words():
    yield "This"
    yield "is"
    yield "a"
    yield "complete"
    yield "sentence"
    # calling next after this yield raises StopIteration


if __name__ == "__main__":
    r = my_range(100, 500, step=100)
    print(r)
    print([num for num in r])

    w = words()
    while True:
        print(next(w))
```

```
$ python3 08-generators/example-04.py
```

```
<generator object my_range at 0x0000000002be4750>
[100, 200, 300, 400]
This
is
a
complete
sentence
Traceback (most recent call last):
  File "/08-generators/example-04.py", line 24, in
<module>
    print(next(w))
StopIteration
```

© 2019 Software Development Academy All Rights Reserved

Advantages of generators

Advantages of using generators:

1. Easier to implement than pure iterators
2. Memory efficient: a list of million elements takes a lot of memory, whereas a generator generating a million elements may take very little
3. They can represent infinite streams
4. They are easily defined by generator expressions

© 2019 Software Development Academy All Rights Reserved

Example 08-generators/example-05.py

```python
def fibonacci():
    """This generator returns consecutive items in Fibonacci sequence.
    Fibonacci sequence is infinite: its first element equals 0, the
    second equals 1, and every following element equals the sum of
    two previous elements:
    0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ..."""
    a = 0
    b = 1
    while True:
        yield a
        a, b = b, a + b


if __name__ == "__main__":
    # Since Fibonacci sequence is infinite, we will print first 10 elements

    for i, element in enumerate(fibonacci()):
        if i > 10:  # this is crucial, otherwise the program runs forever
            break
        print(f"fib({i}): {element}")
```

```
$ python3 08-generators/example-05.py
```

```
fib(0): 0
fib(1): 1
fib(2): 1
fib(3): 2
fib(4): 3
fib(5): 5
fib(6): 8
fib(7): 13
fib(8): 21
fib(9): 34
fib(10): 55
```

© 2019 Software Development Academy All Rights Reserved

Exercise 2

1. Write a generator called `elements` which takes two arguments: generator `gen` and integer `max_elem`. `elements` should yield no more than `max_elem` items from generator `gen`. We assume `gen` is always infinite or longer than `max_elem`.
2. Use `elements` to get 50 first numbers in Fibonacci sequence from *example-05.py*

© 2019 Software Development Academy All Rights Reserved

Generator expressions

1. Generators can be also created using generator expressions
2. Generator expressions look like list comprehensions, but use parentheses instead of square brackets.

© 2019 Software Development Academy All Rights Reserved

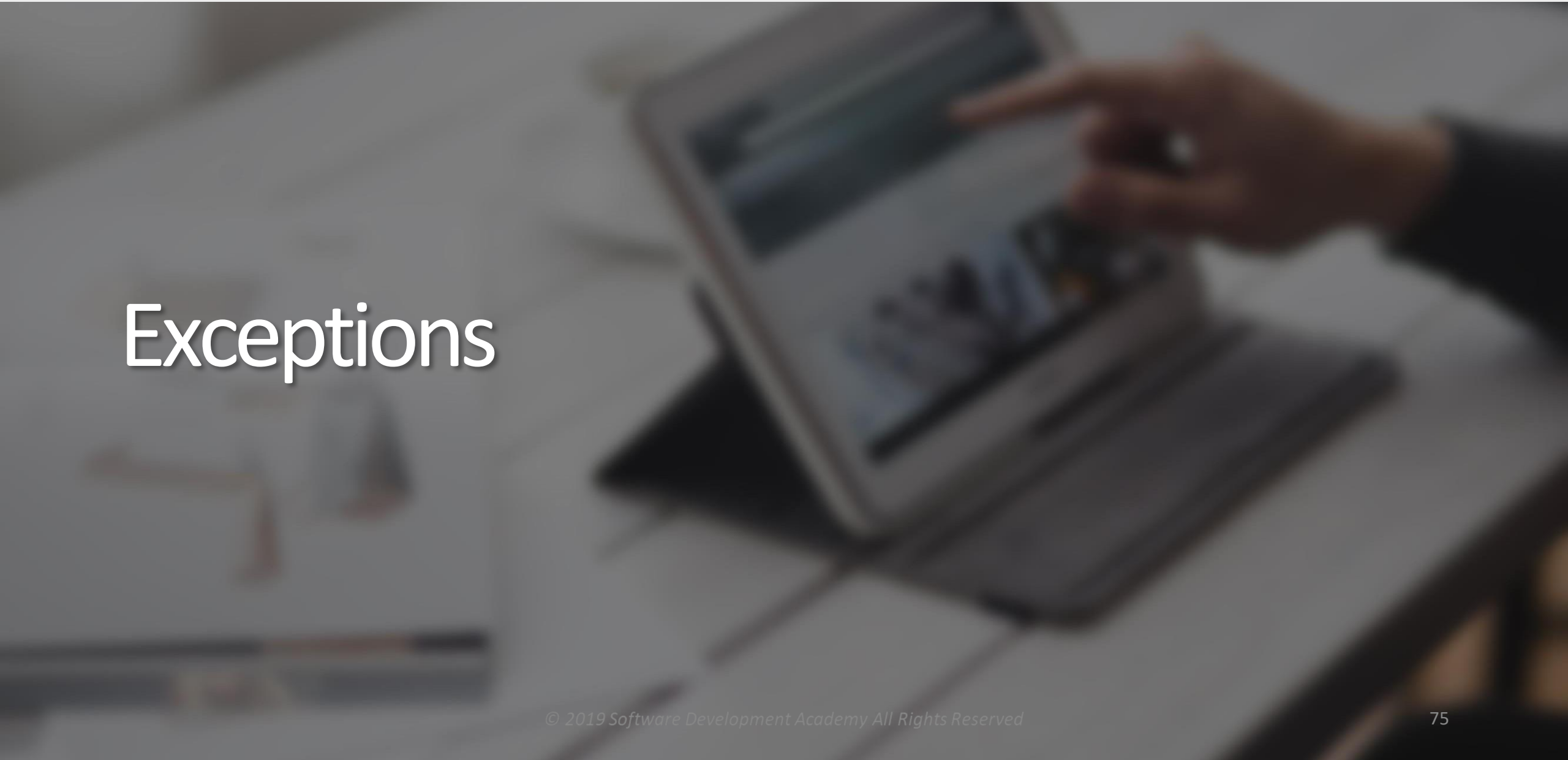
Example 08-generators/example-06.py

```
import time
import itertools

start_time = time.time()
list_of_cubes = [x ** 3 for x in range(1_000_000)]
delta = time.time() - start_time
print(f"Creating a list of 1 000 000 cubes in memory took {delta:03f}s")

start_time = time.time()
generator_of_cubes = (x ** 3 for x in range(1_000_000))
delta = time.time() - start_time
print(f"Creating a generator of 1 000 000 cubes in memory took {delta:03f}s")

print(list_of_cubes[:10])
print(list(itertools.islice(generator_of_cubes, 10)))
```

```
$ python3 08-generators/example-06.py
```

```
Creating a list of 1 000 000 cubes in memory took 0.297460s
Creating a generator of 1 000 000 cubes in memory took 0.000000s
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

© 2019 Software Development Academy All Rights Reserved



Exceptions

© 2019 Software Development Academy All Rights Reserved

Exceptions

1. Whenever something goes irreversibly wrong, Python raises an `Exception`
2. A single exception from that behaviour is `SyntaxError`, which is raised by Python interpreter whenever it cannot interpret the code you have written
3. All of the Exceptions are subclasses of `BaseException` class
4. All user defined Exceptions should inherit from `Exception` class

© 2019 Software Development Academy All Rights Reserved

Types of exceptions

There are many different types of errors in Python, but the most common are:

AssertionError	KeyError	RecursionError
AttributeError	KeyboardInterrupt	ReferenceError
EOFError	MemoryError	RuntimeError
GeneratorExit	NameError	StopIteration
ImportError	NotImplementedError	SystemError
ModuleNotFoundError	OSError	SystemExit
IndexError	OverflowError	TypeError
IOError	EnvironmentError	ZeroDivisionError

Further reading: https://docs.python.org/3/library/exceptions.html#exception-hierarchy

© 2019 Software Development Academy All Rights Reserved

Example 09-exceptions/example-01.py

```
raise Exception("This is a manually raised exception!")
```

```
$ python3 09-exceptions/example-01.py
```

```
Traceback (most recent call last):
  File "/09-exceptions/example-01.py", line 1, in <module>
    raise Exception("This is a manually raised exception!")
Exception: This is a manually raised exception!
```

© 2019 Software Development Academy All Rights Reserved

Exercise 1

1. Try raising some other types of exceptions!

© 2019 Software Development Academy All Rights Reserved

Catching an exception

1. Python supports catching an `Exception` using a `try … except` statement
2. `try … except` is a context manager underneath
3. It executes all the statements in `try` block, unless one of them raises an exception
4. If an exception is raised, execution of try block is stopped and `except` block is executed
5. We can (and should) specify types of Exceptions that should be handled by `except` block
6. You can define multiple `except` blocks for different exception types

© 2019 Software Development Academy All Rights Reserved

Example 09-exceptions/example-02.py

```
try:
    print("First statement")
    print("Second statement")
    raise Exception("Oh no!")
    print("Final statement")
except Exception:
    print("An exception occurred!")


try:
    l = []
    print(l[10])
except IndexError:
    print("IndexError has ocurred!")
```

```
$ python3 09-exceptions/example-02.py
```

```
First statement
Second statement
An exception occurred!
IndexError has ocurred!
```

© 2019 Software Development Academy All Rights Reserved

Exercise 2

1. Given function `random_exception` which raises a random exception every time, write a `try-except` block which handles each of the possible exceptions by counting their occurrences. Print the tally.

© 2019 Software Development Academy All Rights Reserved

Custom exceptions

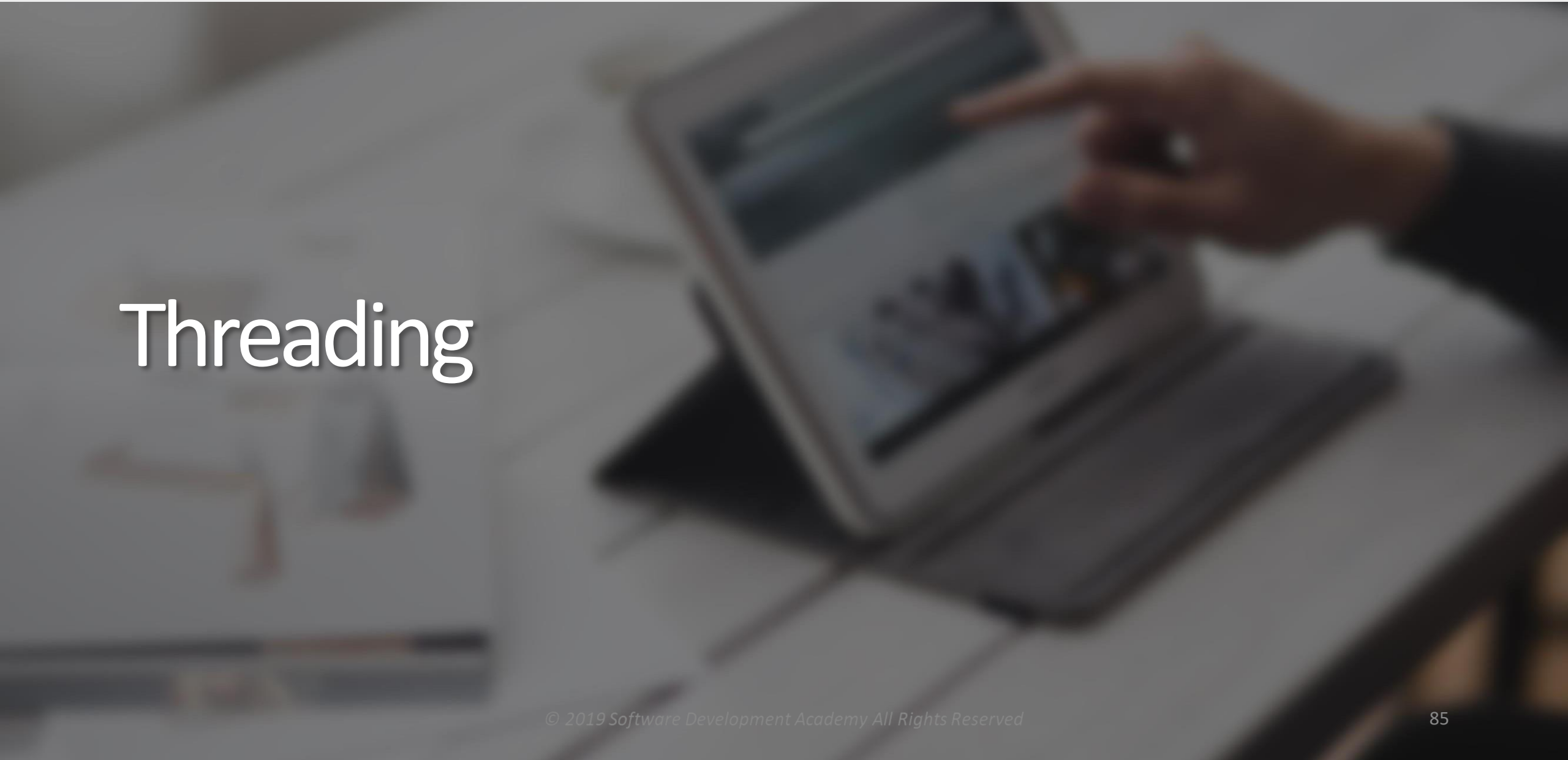
1. Defining custom exceptions is as simple, as creating a new class, which inherits after `Exception`
2. Syntax: `class MyException(Exception): pass`

© 2019 Software Development Academy All Rights Reserved

Exercise 3

1. Define your own exceptions:
   - `PasswordTooShort`
   - `PasswordTooLong`
   - `InvalidPassword`
2. Write a function `validate_password()`:
   - check whether user input collected via `input()` equals `secret_password` variable.
   - If the input is too short (less than 3 characters) or too long (longer than 30 characters), raise appropriate exceptions
   - If the password does not match, raise `InvalidPassword`
3. Handle all exceptions raised in `validate_password` by printing a nice message.

© 2019 Software Development Academy All Rights Reserved



Threading

© 2019 Software Development Academy All Rights Reserved

What is a sequential execution?

1. All the code in your scripts up to this point were executed sequentially
2. This means an instruction had to finish for the next one to start
3. However modern computers can do things *in parallel*
4. This means you can speed up the execution of tasks if the tasks are independent

© 2019 Software Development Academy All Rights Reserved

Example 10-threading/example-01.py

```python
import logging
import time

l = logging.getLogger("toll_booth")
h = logging.StreamHandler()
f = logging.Formatter("%(asctime)s: %(message)s")
h.setFormatter(f)
l.addHandler(h)
l.setLevel(logging.INFO)


def process_toll_booth_fee(car):
    l.info(f"Processing {car}'s fee...")
    time.sleep(2)  # processing takes 2 seconds
    l.info(f"Done processing {car}'s fee.")


if __name__ == "__main__":
    cars = ["Red", "Blue", "Green"]

    start_time = time.time()
    for car in cars:
        process_toll_booth_fee(car)
    delta_time = time.time() - start_time
    print(f"It took {delta_time:.1f}s to process all the cars.")
```

```
$ python3 10-threading/example-01.py
```

```
2019-09-13 18:07:08,764: Processing Red's fee...
2019-09-13 18:07:10,777: Done processing Red's fee.
2019-09-13 18:07:10,777: Processing Blue's fee...
2019-09-13 18:07:12,793: Done processing Blue's fee.
2019-09-13 18:07:12,793: Processing Green's fee...
2019-09-13 18:07:14,808: Done processing Green's fee.
It took 6.0s to process all the cars.
```

© 2019 Software Development Academy All Rights Reserved

What is a thread?

1. Since sequential execution can be slow, we will be using threads
2. Threads are sequences of instructions that can be processed separately by scheduler
3. Since many threads can run concurrently, they speed up the overall execution of code
4. A thread in Python can run when other threads are awaiting some result - like sleeping

© 2019 Software Development Academy All Rights Reserved

Example 10-threading/example-02.py

```python
import logging
import time
import threading

l = logging.getLogger("toll_booth")
h = logging.StreamHandler()
f = logging.Formatter("%(asctime)s: %(message)s")
h.setFormatter(f)
l.addHandler(h)
l.setLevel(logging.INFO)

def process_toll_booth_fee(car):
    l.info(f"Processing {car}'s fee...")
    time.sleep(2)  # processing takes 2 seconds
    l.info(f"Done processing {car}'s fee.")

if __name__ == "__main__":
    cars = ["Red", "Blue", "Green"]
    threads = []

    start_time = time.time()
    for car in cars:
        # launch a new thread for each car
        new_thread = threading.Thread(target=process_toll_booth_fee, args=(car,))
        threads.append(new_thread)
        new_thread.start()

    for t in threads:
        t.join()
    delta_time = time.time() - start_time
    print(f"It took {delta_time:.1f}s to process all the cars.")
```

```
$ python3 10-threading/example-02.py
```

```
2019-09-13 18:09:22,965: Processing Red's fee...
2019-09-13 18:09:22,965: Processing Blue's fee...
2019-09-13 18:09:22,965: Processing Green's fee...
2019-09-13 18:09:24,987: Done processing Red's fee.
2019-09-13 18:09:24,987: Done processing Blue's fee.
2019-09-13 18:09:24,987: Done processing Green's fee.
It took 2.0s to process all the cars.
```

© 2019 Software Development Academy All Rights Reserved

Function explanation

1. This time it took ~2 seconds for all the tasks to finish, because they were running in concurrently
2. `threading.Thread` creates a new thread using function supplied in `target` and arguments in `args`
3. `.start()` starts a thread
4. `.join()` makes sure that the main thread - the one the application is running in, awaits thread completion

© 2019 Software Development Academy All Rights Reserved

Exercise 1

1. Remove `.join()` statements from *example-02.py*. What happens?

© 2019 Software Development Academy All Rights Reserved

Daemon threads

1. Lifetime of daemon threads is strictly tied to the lifetime of an application which started them
2. If an application exits, the threads are killed, without a chance to finish
3. You can spawn a daemonic thread by adding `daemon=True` to `threading.Thread` constructor

© 2019 Software Development Academy All Rights Reserved

Exercise 2

1. Make all the threads in Exercise 1 daemonic. What happens when you run the application?

© 2019 Software Development Academy All Rights Reserved

Thread pool

1. Now that you know how to launch threads and wait for them to finish the hard way, you can use the easy way
2. `concurrent.futures` module contains a `ThreadPool` which does just that
3. `ThreadPool` is a context manager
4. It spawns a pool of maximum `max_workers`, mapping the arguments to the callable
5. At exit, it waits for all the threads to finish

© 2019 Software Development Academy All Rights Reserved

Example 10-threading/example-03.py

```python
import logging
import time
import concurrent.futures

l = logging.getLogger("toll_booth")
h = logging.StreamHandler()
f = logging.Formatter("%(asctime)s: %(message)s")
h.setFormatter(f)
l.addHandler(h)
l.setLevel(logging.INFO)


def process_toll_booth_fee(car):
    l.info(f"Processing {car}'s fee...")
    time.sleep(2)  # processing takes 2 seconds
    l.info(f"Done processing {car}'s fee.")


if __name__ == "__main__":
    cars = ["Red", "Blue", "Green"]

    start_time = time.time()
    # Note that we allow max 2 workers, so Green will be processed after one
    # of the previous cars finishes processing its payment.
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
        executor.map(process_toll_booth_fee, cars)
    delta_time = time.time() - start_time
    print(f"It took {delta_time:.1f}s to process all the cars.")
```

```
$ python3 10-threading/example-03.py
```

```
2019-09-13 18:13:40,508: Processing Red's fee...
2019-09-13 18:13:40,508: Processing Blue's fee...
2019-09-13 18:13:42,513: Done processing Blue's fee.
2019-09-13 18:13:42,513: Done processing Red's fee.
2019-09-13 18:13:42,513: Processing Green's fee...
2019-09-13 18:13:44,514: Done processing Green's fee.
It took 4.1s to process all the cars.
```

© 2019 Software Development Academy All Rights Reserved

Race conditions & shared memory

1. Since all threads share memory, they are free to modify it
2. However, since they are running concurrently, there is a risk of *race condition*
3. A race condition is a hazard of modifying a memory in an unexpected order, causing an error to occur
4. Let's take a look at *example-04.py*

© 2019 Software Development Academy All Rights Reserved

Example 10-threading/example-04.py

```python
import logging
import time
import concurrent.futures

l = logging.getLogger("toll_booth")
(…)
l.setLevel(logging.INFO)

class TollBooth:
    def __init__(self):
        self.register = 0.0

    def process_fee(self, car, fee):
        l.info(f"Processing {car}'s fee. Current total: {self.register}")
        new_total = self.register + fee  # Toll booth calculates a new total
        time.sleep(0.1)  # processing takes 0.1 seconds
        self.register = new_total  # New total is saved after 0.1 seconds
        l.info(f"Done processing {car}'s fee. New total: {self.register}")

if __name__ == "__main__":
    cars = ["Red", "Blue", "Green"]
    booth = TollBooth()
    start_time = time.time()
    with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
        for c in cars:
            executor.submit(booth.process_fee, c, 10.0)
    delta_time = time.time() - start_time
    print(f"It took {delta_time:.1f}s to process all the cars.")
    print(f"Total: {booth.register}")
```

```
$ python3 10-threading/example-04.py
```

```
2019-09-13 18:15:40,394: Processing Red's fee. Current total: 0.0
2019-09-13 18:15:40,394: Processing Blue's fee. Current total: 0.0
2019-09-13 18:15:40,394: Processing Green's fee. Current total: 0.0
2019-09-13 18:15:40,510: Done processing Red's fee. New total: 10.0
2019-09-13 18:15:40,510: Done processing Blue's fee. New total: 10.0
2019-09-13 18:15:40,510: Done processing Green's fee. New total: 10.0
It took 0.1s to process all the cars.
Total: 10.0
```

© 2019 Software Development Academy All Rights Reserved

Explanation of Example 4

1. Each of the threads started processing simultaneously
2. They managed to access the same initial value and each of them saved the same increased value to the register
3. Hence after processing 3 payments, the register indicated only 10.0 instead of 30.0

© 2019 Software Development Academy All Rights Reserved

Locks

1. Locks are one of the solutions to shared memory issue
2. When a lock is *acquired* by a thread, no other thread can acquire the lock
3. This enables thread to safely modify shared memory
4. When the thread is finished, it *releases* the lock
5. The lock now can be acquired by another thread
6. This kind of mutually exclusive lock is called a *mutex*
7. The section between *acquiring* and *releasing* the lock is called *critical section*

© 2019 Software Development Academy All Rights Reserved

Example 10-threading/example-05.py

```python
import logging
import time
import concurrent.futures
import threading

l = logging.getLogger("toll_booth")
# (…)
class TollBooth:
    def __init__(self):
        self.register = 0.0
        self.__lock = threading.Lock()

    def process_fee(self, car, fee):
        l.info(f"Processing {car}'s fee. Current total: {self.register}")
        with self.__lock:
            new_total = self.register + fee  # Toll booth calculates a new total
            time.sleep(0.1)  # processing takes 0.1 seconds
            self.register = new_total  # New total is saved after 0.1 seconds
        # notice operations outside the critical section are still concurrent
        time.sleep(1)
        l.info(f"Done processing {car}'s fee. New total: {self.register}")

if __name__ == "__main__":
    cars = ["Red", "Blue", "Green"]
    booth = TollBooth()

    start_time = time.time()
    with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
        for c in cars:
            executor.submit(booth.process_fee, c, 10.0)
    delta_time = time.time() - start_time
    print(f"It took {delta_time:.1f}s to process all the cars.")
    print(f"Total: {booth.register}")
```

```
$ python3 10-threading/example-05.py
```

```
2019-09-13 18:18:12,770: Processing Red's fee. Current total: 0.0
2019-09-13 18:18:12,770: Processing Blue's fee. Current total: 0.0
2019-09-13 18:18:12,770: Processing Green's fee. Current total: 0.0
2019-09-13 18:18:13,905: Done processing Red's fee. New total: 30.0
2019-09-13 18:18:14,007: Done processing Blue's fee. New total: 30.0
2019-09-13 18:18:14,123: Done processing Green's fee. New total: 30.0
It took 1.4s to process all the cars.
Total: 30.0
```

© 2019 Software Development Academy All Rights Reserved

Exercise 3

1. Write a class Bakery, which:
   - has a common variable called `storage` (int)
   - has a function called `baker()` which (for 10 loops) notes down current state of the `storage`, sleeps for 0.1s and adds 1 to the `storage`, then sleeps for 1s
   - has a function called `customer()` which (for 10 loops) notes down current state of the `storage`, sleeps for 0.2s and removes 2 from the `storage` `if storage >= 2`, then sleeps for 1s
2. use `threading.Lock` to prevent race conditions
3. Run 2 customer threads and 1 baker thread
4. Use `logging` to ensure all functions are working

© 2019 Software Development Academy All Rights Reserved

Queues

1. Exercise 3 has been a perfect example of Producer-Consumer pattern
2. There is however a better way of distributing data in such a pattern
3. Enter a `Queue`
4. A `Queue` (on the surface) is a collection like list or dictionary, but it is thread-safe
5. This means we can use it without using locks
6. `Queue` is FIFO by default, which means the first item added to the queue will be the first to leave it (first in, first out)

© 2019 Software Development Academy All Rights Reserved

Example 10-threading/example-06.py

```python
import logging
import time
import concurrent.futures
import queue
import random

l = logging.getLogger("bakery")
# (…)

class Bakery:
    def __init__(self):
        self.storage = queue.Queue()

    def baker(self):
        for i in range(12):
            self.storage.put(f"Bread #{i}")
            l.info("Finished baking")
            time.sleep(0.2)

    def customer(self, i):
        for _ in range(3):
            bread = self.storage.get()
            l.info(f"Customer #{i} got {bread}")
            time.sleep(random.uniform(0.5, 2.0))  # notice the random frequency


if __name__ == "__main__":
    mine = Bakery()
    with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
        executor.submit(mine.baker)
        executor.submit(mine.customer, 1)
        executor.submit(mine.customer, 2)
        executor.submit(mine.customer, 3)
        executor.submit(mine.customer, 4)
```

```
$ python3 10-threading/example-06.py
```

```
2019-09-13 18:22:35,436: Finished baking
2019-09-13 18:22:35,436: Customer #1 got Bread #0
2019-09-13 18:22:35,647: Finished baking
2019-09-13 18:22:35,647: Customer #2 got Bread #1
2019-09-13 18:22:35,861: Finished baking
2019-09-13 18:22:35,861: Customer #3 got Bread #2
2019-09-13 18:22:36,062: Finished baking
2019-09-13 18:22:36,062: Customer #4 got Bread #3
2019-09-13 18:22:36,263: Finished baking
2019-09-13 18:22:36,470: Finished baking
2019-09-13 18:22:36,671: Finished baking
2019-09-13 18:22:36,877: Finished baking
2019-09-13 18:22:36,954: Customer #3 got Bread #4
2019-09-13 18:22:37,053: Customer #1 got Bread #5
2019-09-13 18:22:37,084: Customer #4 got Bread #6
2019-09-13 18:22:37,084: Finished baking
2019-09-13 18:22:37,297: Finished baking
2019-09-13 18:22:37,451: Customer #2 got Bread #7
2019-09-13 18:22:37,497: Finished baking
2019-09-13 18:22:37,703: Finished baking
2019-09-13 18:22:38,109: Customer #1 got Bread #8
2019-09-13 18:22:38,341: Customer #4 got Bread #9
2019-09-13 18:22:38,446: Customer #2 got Bread #10
2019-09-13 18:22:38,699: Customer #3 got Bread #11
```

© 2019 Software Development Academy All Rights Reserved

Sockets

1. Sockets are a low-level way of communicating over the network.
2. A socket can either listen on a specific *port* or send data to a port
3. We'll try communicating two threads using sockets
4. Let's write a simple server and client
5. A client will send strings provided via `input()` to the server
6. The server will respond with the same string, prefixed by 'Server:`
7. We will allow maximum of 3 such messages

© 2019 Software Development Academy All Rights Reserved

Example 10-threading/example-07.py

```python
import socket
import sys
import threading
import logging

l = logging.getLogger("sockets")
# …

def server(listen_on_port):
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    except socket.error:
        l.critical("Error creating socket server")
        sys.exit(1)
    sock.bind(("0.0.0.0", listen_on_port))
    sock.listen(1)
    l.info("Socket server is up")
    connection, _ = sock.accept()
    for _ in range(3):
        data = connection.recv(1024)  # 1024 is max. buffer length
        connection.sendall(f"Server: {data.decode()}".encode())
    connection.close()
    l.info("Closing socket server")
    sock.close()
```

```python
def client(send_to_port):
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    except socket.error:
        l.critical("Error creating socket server")
        sys.exit(1)
    l.info("Client is active")
    sock.connect(("localhost", send_to_port))
    for _ in range(3):
        sock.sendall(input("Message: ").encode())
        print(sock.recv(1024).decode())
    sock.close()
    l.info("Closing client")

if __name__ == "__main__":
    port = 8081
    srv = threading.Thread(target=server, args=(port,))
    srv.start()
    client(port)
    srv.join()
```

```
$ python3 10-threading/example-07.py
```

```
2019-09-13 18:26:23,277: Client is active
2019-09-13 18:26:23,277: Socket server is up
Message: Hello
Server: Hello
Message: There
Server: There
Message: Is that server?
Server: Is that server?
2019-09-13 18:26:31,252: Closing socket server
2019-09-13 18:26:31,253: Closing client
```

© 2019 Software Development Academy All Rights Reserved

Threading vs Multiprocessing

1. Now that you know threads, you feel ready to writing code running concurrently
2. However there is a difference between concurrency and parallelism
3. Switches between threads happen when they are awaiting some external operation (like sleep, calling a database, sending a web request and so on) to finish
4. This is why threading speeds up I/O (input-output) bound operations.
5. While one of the threads is waiting for the I/O operation to finish, another takes its place and executes until it too blocks, or it is preempted by the CPU.
6. GIL (Python's Global Interpreter Lock) ensures that no more than one thread executes at the same time.
7. The true parallelism is achieved by using `multiprocessing` library, which can utilize multiple CPU cores.
8. However multiprocessing is best for CPU-bound problems
9. CPU-bound problems are the ones that are CPU heavy - like calculating a square root of a gigantic number or processing a list of five million e-mails.

© 2019 Software Development Academy All Rights Reserved

Multiprocessing

© 2019 Software Development Academy All Rights Reserved

Multiprocessing

1. Just like in case of `Threads`, you can spawn multiple processes by creating `Process` class instances
2. Start them with `.start()`
3. And wait for them to be done using `.join()`

© 2019 Software Development Academy All Rights Reserved

Example 11-multiprocessing/example-01.py

```python
import logging
import time
import multiprocessing

l = logging.getLogger("multiprocessing")
h = logging.StreamHandler()
f = logging.Formatter("%(asctime)s: %(message)s")
h.setFormatter(f)
l.addHandler(h)
l.setLevel(logging.INFO)


def say_hello(i):
    l.info(f"Hello from #{i}")
    time.sleep(1)


if __name__ == "__main__":
    processes = [
        multiprocessing.Process(target=say_hello, args=(i + 1,)) for i in range(5)
    ]
    start_time = time.time()
    for p in processes:
        p.start()
    for p in processes:
        p.join()
    delta_time = time.time() - start_time
    print(f"It took {delta_time:.1f}s to say hello.")
```

```
$ python3 11-multiprocessing/example-01.py
```

```
2019-09-13 18:30:59,519: Hello from #1
2019-09-13 18:30:59,534: Hello from #2
2019-09-13 18:30:59,534: Hello from #3
2019-09-13 18:30:59,534: Hello from #4
2019-09-13 18:30:59,551: Hello from #5
It took 1.3s to say hello.
```

© 2019 Software Development Academy All Rights Reserved

Multiprocessing

As stated before, multiprocessing shines the most when the problem is CPU heavy.

© 2019 Software Development Academy All Rights Reserved

Example 11-multiprocessing/example-02.py

```python
import logging
import time
import multiprocessing
import math

l = logging.getLogger("multiprocessing")
# (…)

def square_root(num):
    for _ in range(10_000_000):
        result = math.sqrt(num)
    l.info(f"sqrt of {num} is {result:.1f}")


if __name__ == "__main__":
    big_float = 123_456_789
    processes = [
        multiprocessing.Process(target=square_root, args=(big_float,)) for i in range(5)
    ]
    start_time = time.time()
    for p in processes:
        p.start()
    for p in processes:
        p.join()
    delta_time = time.time() - start_time
    print(f"It took {delta_time:.1f}s to calculate square roots.")
```

```
$ python3 11-multiprocessing/example-02.py
```

```
2019-09-13 18:32:24,080: sqrt of 123456789 is 11111.1
2019-09-13 18:32:24,094: sqrt of 123456789 is 11111.1
2019-09-13 18:32:24,102: sqrt of 123456789 is 11111.1
2019-09-13 18:32:24,176: sqrt of 123456789 is 11111.1
2019-09-13 18:32:24,270: sqrt of 123456789 is 11111.1
It took 2.2s to calculate square roots.
```

© 2019 Software Development Academy All Rights Reserved

Process synchronization

1. Processes can use synchronization primitives like `Lock`, `Queue` etc.
2. The primitives are all part of multiprocessing library
3. They are mostly equivalent to the ones found in threading

© 2019 Software Development Academy All Rights Reserved

Multiprocessing – exercise 1

1. Create the following functions:
    1. `generate_random(out_q, n)` which generates `n` random integers and puts them in `out_q` Queue
    2. `square(in_q, out_q, n)` which takes numbers from the `in_q` Queue and puts their squares in the `out_q`.
    3. `to_binary(in_q, out_q, n)` takes numbers from the `in_q` Queue and puts their binary representations in the `out_q`.
2. Create the three queues needed to connect functions in the following order:
    1. `generate_random`
    2. `queue_a`
    3. `square`
    4. `queue_b`
    5. `to_binary`
    6. `result_q`
3. Run the three functions in separate processes. Make `generate_random` generate 1000 integers.
4. **Important!** Read all the items from `result_q` before calling `.join()`
5. You can add `timeout=<seconds>` to `put/get` calls to avoid deadlocks.

© 2019 Software Development Academy All Rights Reserved

Pipes

1. Pipes are unique to multiprocessing
2. They provide a bi-directional parent-child process communication

© 2019 Software Development Academy All Rights Reserved

Example 11-multiprocessing/example-03.py

```
import multiprocessing


def echo(pipe):
    query = pipe.recv()
    while query is not None:
        pipe.send(f"Echo: {query}")
        query = pipe.recv()


if __name__ == "__main__":
    parent_conn, child_conn = multiprocessing.Pipe()
    proc = multiprocessing.Process(target=echo, args=(child_conn,))
    proc.start()
    parent_conn.send("Hello")
    print(parent_conn.recv())
    parent_conn.send("Hello!")
    print(parent_conn.recv())
    parent_conn.send(None)
    proc.join()
```

```
$ python3 11-multiprocessing/example-03.py
```

```
Echo: Hello
Echo: Hello!
```

© 2019 Software Development Academy All Rights Reserved

Pools

1. Multiprocessing pools are also supported

© 2019 Software Development Academy All Rights Reserved

Example 11-multiprocessing/example-04.py

```python
import random
import multiprocessing


def complex_calculation(a, b, c):
    return a ** b % c


if __name__ == "__main__":
    argument_lists = [
        (random.randint(1, 20), random.randint(1, 20), random.randint(1, 20))
        for _ in range(500)
    ]

    with multiprocessing.Pool(10) as workers:
        result = workers.starmap(complex_calculation, argument_lists)

    for i in range(5):
        print(f"{argument_lists[i]} -> {result[i]}")
    print("...")
```

```
$ python3 11-multiprocessing/example-04.py
```

```
(12, 4, 14) -> 2
(18, 20, 20) -> 16
(2, 19, 4) -> 0
(5, 3, 18) -> 17
(18, 14, 6) -> 0
...
```

© 2019 Software Development Academy All Rights Reserved

Multiprocessing – exercise 2

1. *11-multiprocessing/request.py* shows you how to perform a http request and print out its response (`pip install requests` might be necessary)
2. Call the following list of URLs using:
   - Thread pool, where a single request happens in a single thread
   - Process pool, where a single request happens in a single process
3. Which one is quicker?
4. URL list:
   1. https://google.com
   2. https://amazon.com
   3. https://ebay.com
   4. https://bbc.co.uk
   5. https://youtube.com
   6. https://wikipedia.org
   7. https://twitter.com
   8. https://spotify.com
   9. https://uber.com
   10. https://apple.com
   11. https://microsoft.com

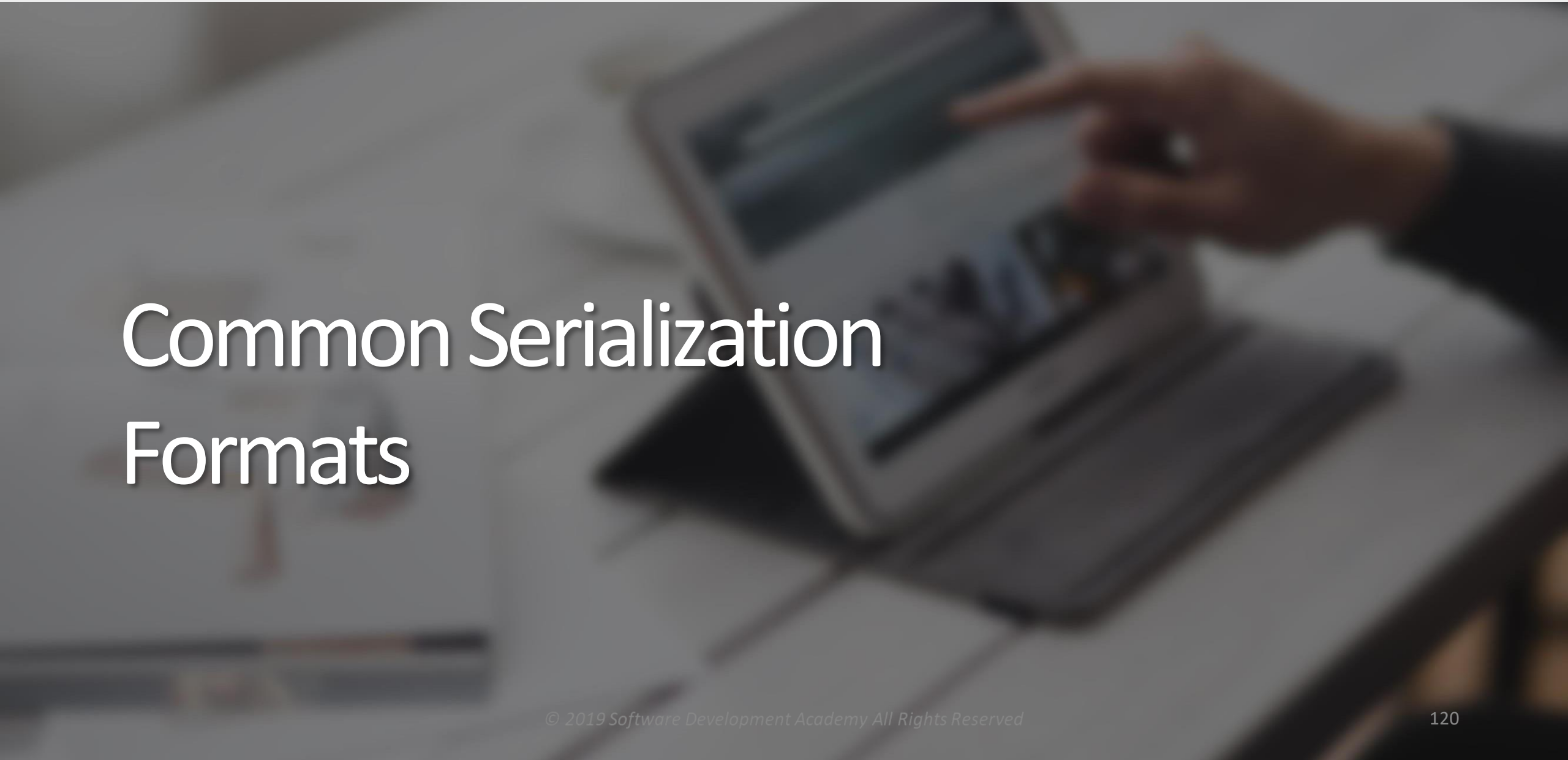
© 2019 Software Development Academy All Rights Reserved

Multiprocessing – exercise 3

Modify multiprocessing solution of exercise 2 & example 3 so that:

1. Main proces creates a Pipe called `urls_parent, urls_child`
2. Exactly 1 child proces starts, listening on the pipe:
    1. If it gets None, it stops execution
    2. If it gets an URL, it sends makes a request prints the time it took to make a request to it

© 2019 Software Development Academy All Rights Reserved



Common Serialization Formats

© 2019 Software Development Academy All Rights Reserved

Common formats

Sometimes we need to save the data to a file. This is called serialization of the data. As the data can be quite complex, it is a good idea to use well defined and commonly used for this task. There are many others, but we chose CSV and JSON to show you the different flavours.

```
id,first_name,last_name,gender,email,job_title
1,Billi,Sutheran,Female,bsutheran0@exblog.jp,Budget/Accounting Analyst I
2,Gwyn,Muschette,Female,gmuschette1@indiegogo.com,Clinical Specialist
3,Jacquelyn,Jerdein,Female,jjerdein2@slate.com,Database Administrator III
4,Dietrich,Pirouet,Male,dpirouet3@ucla.edu,Administrative Officer
5,Elnora,Graver,Female,egraver4@craigslist.org,GIS Technical Architect
```

CSV

```
[{
    "id": 1,
    "first_name": "Jenn",
    "last_name": "Wince",
    "gender": "Female",
    "email": "jwince0@mediafire.com",
    "job_title": "Recruiting Manager"
}, (...)]
```

JSON

© 2019 Software Development Academy All Rights Reserved

CSV

1. CSV stands for „comma-separated values”
2. This format usually contains tabularized data, with first row being headers
3. Columns are separated using a single comma
4. If a column contains a comma, its contents are surrounded with quotes
5. Python provides a ready `csv` library for handling CSV files

```
id,first_name,last_name,gender,email,job_title
1,Billi,Sutheran,Female,bsutheran0@exblog.jp,Budget/Accounting Analyst I
```

© 2019 Software Development Academy All Rights Reserved

Example 12-serialization/example-01.py

```
import csv


if __name__ == "__main__":
    rows = []
    with open("12-serialization/data/example.csv") as f:
        reader = csv.reader(f)
        for row in reader:
            rows.append(row)
            print(row)
    rows.append([6, "Phillis", "Late", "Female", "plate0@loc.gov", "Civil Engineer"])
    with open("12-serialization/data/example.csv", "w+") as f:
        writer = csv.writer(f, lineterminator="\n")
        writer.writerows(rows)
```

```
$ python3 12-serialization/example-01.py
```

```
['id', 'first_name', 'last_name', 'gender', 'email', 'job_title']
['1', 'Billi', 'Sutheran', 'Female', 'bsutheran0@exblog.jp', 'Budget/Accounting Analyst I']
['2', 'Gwyn', 'Muschette', 'Female', 'gmuschette1@indiegogo.com', 'Clinical Specialist']
['3', 'Jacquelyn', 'Jerdein', 'Female', 'jjerdein2@slate.com', 'Database Administrator III']
['4', 'Dietrich', 'Pirouet', 'Male', 'dpirouet3@ucla.edu', 'Administrative Officer']
['5', 'Elnora', 'Graver', 'Female', 'egraver4@craigslist.org', 'GIS Technical Architect']
```

© 2019 Software Development Academy All Rights Reserved

Example 12-serialization/example-02.py

```
import csv


if __name__ == "__main__":
    with open("12-serialization/data/example.csv") as f:
        reader = csv.DictReader(f)
        for row in reader:
            print(dict(row))
```

```
$ python3 12-serialization/example-02.py
```

```
{'id': '1', 'first_name': 'Billi', 'last_name': 'Sutheran', 'gender': 'Female', 'email': 'bsutheran0@exblog.jp',
'job_title': 'Budget/Accounting Analyst I'}
{'id': '2', 'first_name': 'Gwyn', 'last_name': 'Muschette', 'gender': 'Female', 'email':
'gmuschette1@indiegogo.com', 'job_title': 'Clinical Specialist'}
{'id': '3', 'first_name': 'Jacquelyn', 'last_name': 'Jerdein', 'gender': 'Female', 'email': 'jjerdein2@slate.com',
'job_title': 'Database Administrator III'}
{'id': '4', 'first_name': 'Dietrich', 'last_name': 'Pirouet', 'gender': 'Male', 'email': 'dpirouet3@ucla.edu',
'job_title': 'Administrative Officer'}
{'id': '5', 'first_name': 'Elnora', 'last_name': 'Graver', 'gender': 'Female', 'email': 'egraver4@craigslist.org',
'job_title': 'GIS Technical Architect'}
```

© 2019 Software Development Academy All Rights Reserved

Exercise 1

1. Using either a reader or a DictReader, load records from 12-serialization/data/people.csv file
2. Print the following information:
    1. Number of all records
    2. Number of people missing an e-mail address
    3. Number of people with middle name
3. Save all the records with e-mail missing to a new file

© 2019 Software Development Academy All Rights Reserved

JSON

1. CSV stands for „comma-separated values”
2. This format usually contains tabularized data, with first row being headers
3. Columns are separated using a single comma
4. If a column contains a comma, its contents are surrounded with quotes
5. Python provides a ready `csv` library for handling CSV files

```
id,first_name,last_name,gender,email,job_title
1,Billi,Sutheran,Female,bsutheran0@exblog.jp,Budget/Accounting Analyst I
```

© 2019 Software Development Academy All Rights Reserved

JSON

1. JSON stands for „JavaScript Object Notation”
2. It’s a format commonly used in web development
3. Python provides a ready `json` library for handling JSON files

```
{
    "id": 1,
    "first_name": "Jenn",
    "last_name": "Wince",
    "gender": "Female",
    "email": "jwince0@mediafire.com",
    "job_title": "Recruiting Manager"
}
```

© 2019 Software Development Academy All Rights Reserved

Example 12-serialization/example-03.py

```python
import json


if __name__ == "__main__":
    with open("12-serialization/data/example.json") as f:
        data = json.load(f)
    for obj in data:
        print(obj)

    data.append(
        {
            "id": "6",
            "first_name": "Phillis",
            "last_name": "Late",
            "gender": "Female",
            "email": "plate0@loc.gov",
            "job_title": "Civil Engineer",
        }
    )
    with open("12-serialization/data/example.json", "w+") as f:
        json.dump(data, f)
```

```
$ python3 12-serialization/example-03.py
```

```
{'id': 1, 'first_name': 'Jenn', 'last_name': 'Wince', 'gender': 'Female', 'email': 'jwince0@mediafire.com', 'job_title': 'Recruiting Manager'}
{'id': 2, 'first_name': 'Finn', 'last_name': 'Cucuzza', 'gender': 'Male', 'email': 'fcucuzza1@ucsd.edu', 'job_title': 'VP Accounting'}
{'id': 3, 'first_name': 'Harriett', 'last_name': 'Durdan', 'gender': 'Female', 'email': 'hdurdan2@alexa.com', 'job_title': 'Office Assistant II'}
{'id': 4, 'first_name': 'Danella', 'last_name': 'Shirtliff', 'gender': 'Female', 'email': 'dshirtliff3@youtube.com', 'job_title': 'Physical Therapy
Assistant'}
{'id': 5, 'first_name': 'Maren', 'last_name': "O'Brien", 'gender': 'Female', 'email': 'mobrien4@desdev.cn', 'job_title': 'Internal Auditor'}
```

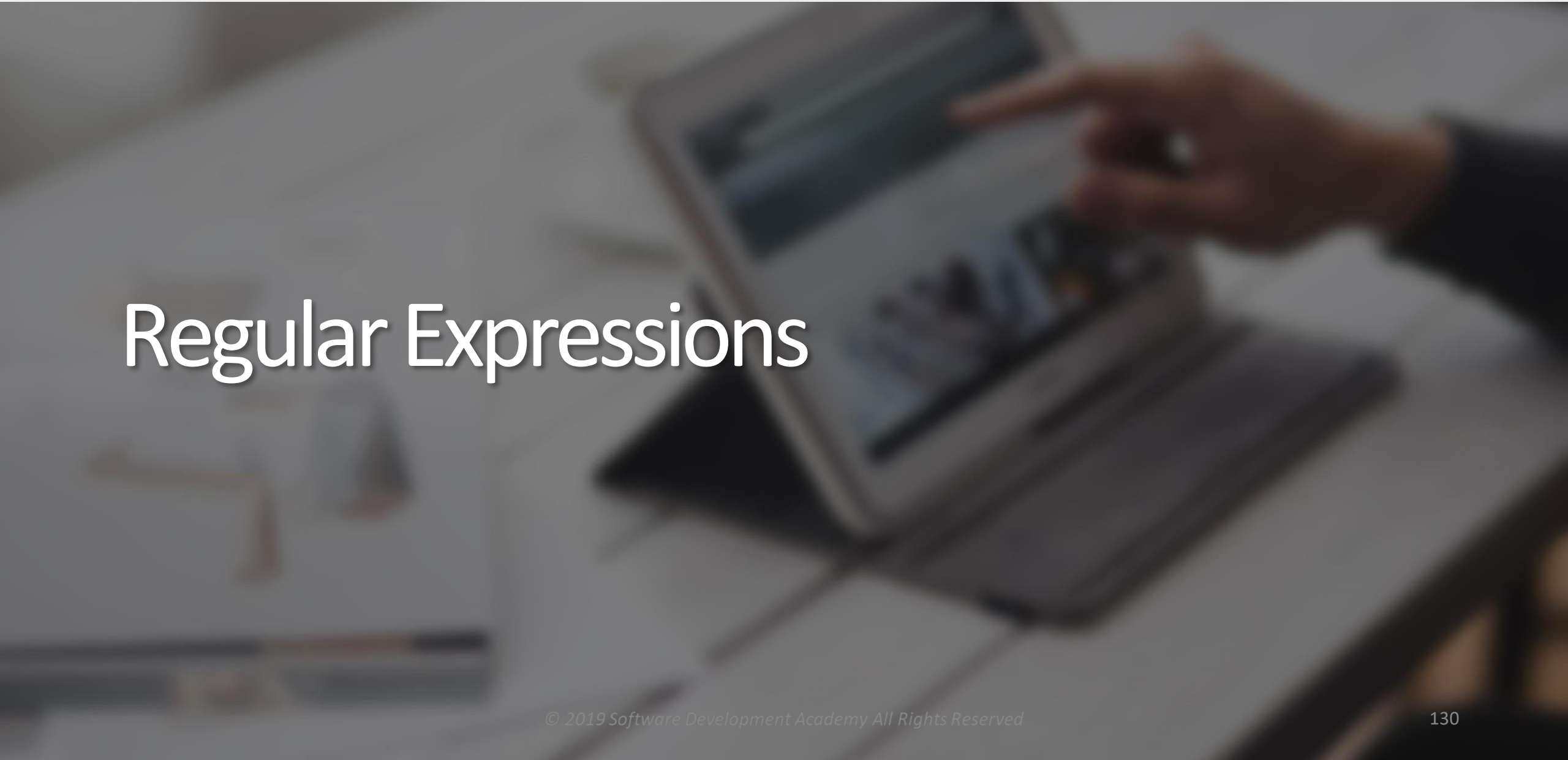
© 2019 Software Development Academy All Rights Reserved

Exercise 2

1. Knowing that json.loads() loads JSON from string instead of a file, load given JSON string and save it as CSV and JSON files.

```
[
 {"id":1,"lat":16.3112941,"long":104.8221147,"country":"Thailand","city":"Don Tan"},
 {"id":2,"lat":41.400351,"long":-8.5116191,"country":"Portugal","city":"Antas"}
]
```

© 2019 Software Development Academy All Rights Reserved



Regular Expressions

© 2019 Software Development Academy All Rights Reserved

Regular expressions

1. Regular expressions are a powerful tool used to serarch, match and replace text
2. Regular expressions allow you to define rules used to perform those operations on a piece of text
3. This functionality is provided by a built-in `re` module in Python

© 2019 Software Development Academy All Rights Reserved

Example 12-regex/example-01.py

```
import re

title = '"The Adventures of Sherlock Holmes" by Arthur Conan Doyle'
for pattern in ["by", "by ......", "by .+", "bygone"]:
    match = re.search(pattern, title)
    if match:
        print(match.group())
    else:
        print(f"No match for '{pattern}'")
```

```
$ python3 13-regex/example-01.py
```

```
by
by Arthur
by Arthur Conan Doyle
No match for 'bygone'
```

© 2019 Software Development Academy All Rights Reserved

Regexp elements

1. A-Z, a-z, 0-9 – ordinary characters
2. . – period matches any character except newline
3. \w – matches a „word” character
4. \s – matches whitespace characters
5. \t, \n – tab and newline
6. \d – matches digits 0-9
7. ^ - matches start of the string
8. $ - matches end of the string
9. \ - escapes special characters, e.g. to match $, use \$

© 2019 Software Development Academy All Rights Reserved

Exercise 1

1. Write regular expression which finds matches the first three word characters of a line
   Test it with:

```
I.     A Scandal in Bohemia
II.    The Red-Headed League
III.   A Case of Identity
IV.    The Boscombe Valley Mystery
V.     The Five Orange Pips
```

© 2019 Software Development Academy All Rights Reserved

Pattern compilation

1. Patterns take a lot of compute time to proces
2. Whenever you are reusing a pattern, you should compile it first
3. Syntax: `pattern = re.compile(pattern_string)`

© 2019 Software Development Academy All Rights Reserved

Example 12-regex/example-02.py

```
import re

lines = [
    "I.     A Scandal in Bohemia",
    "II.    The Red-Headed League",
    "III.   A Case of Identity",
    "IV.    The Boscombe Valley Mystery",
    "V.     The Five Orange Pips",
]
pattern = re.compile("\w+\.")
for l in lines:
    match = re.search(pattern, l)
    if match:
        print(match.group())
```

```
$ python3 13-regex/example-02.py
```

```
I.
II.
III.
IV.
V.
```

© 2019 Software Development Academy All Rights Reserved

Symbol repetition

1. You can use `<symbol>*` to define a symbol which occurs **zero** or more times
2. You can use `<symbol>+` to define a symbol which occurs **one** or more times
3. You can use `<symbol>{min,max}` to define a symbol which occurs between `min` and `max` times

© 2019 Software Development Academy All Rights Reserved

Example 12-regex/example-03.py

```
import re

lines = [
    "I.     A Scandal in Bohemia",
    "II.    The Red-Headed League",
    "III.   A Case of Identity",
    "IV.    The Boscombe Valley Mystery",
    "V.     The Five Orange Pips",
]
pattern = re.compile("^\w+\.\s+.*$")
for l in lines:
    match = re.search(pattern, l)
    if match:
        print(match.group())
```

```
$ python3 13-regex/example-03.py
```

```
I.     A Scandal in Bohemia
II.    The Red-Headed League
III.   A Case of Identity
IV.    The Boscombe Valley Mystery
V.     The Five Orange Pips
```

© 2019 Software Development Academy All Rights Reserved

Exercise 2

UUID (universally unique identifier) format is defined as:

- 5 groups of lowercase letters and digits
- Separated with dashes
- Group lengths: 8-4-4-4-12

Write a pattern which matches the UUID format and test it with:

```
377b1a2e-79c1-4c32-9ef2-92389f16f0de
3e233770-9f67-4999-9944-563ce50f1678
626ce1cd-4441-4957-bbae-5582096cf62d
```

© 2019 Software Development Academy All Rights Reserved

Brackets

1. Sometimes we need to allow more than one type to be matched
2. This can be done using square brackets: `[<symbols>]`
3. Any of the symbols inside the brackets can be matched succesfully
4. Brackets support symbol repetition, like `[<symbols>]+`

© 2019 Software Development Academy All Rights Reserved

Example 12-regex/example-04.py

```
import re

lines = [
    "I.     A Scandal in Bohemia",
    "II.    The Red-Headed League",
    "III.   A Case of Identity",
    "IV.    The Boscombe Valley Mystery",
    "V.     The Five Orange Pips",
]
pattern = re.compile("[IV]+\.\s+[A-Za-z ]{1,200}")
for l in lines:
    match = re.search(pattern, l)
    if match:
        print(match.group())
```

```
$ python3 13-regex/example-04.py
```

```
I.     A Scandal in Bohemia
II.    The Red-Headed League
III.   A Case of Identity
IV.    The Boscombe Valley Mystery
V.     The Five Orange Pips
```

© 2019 Software Development Academy All Rights Reserved

Exercise 3

Write a pattern matching the following e-mails. Test it.

```
bjandourek0@stanford.edu
mtimothy1@deviantart.com
mdodding2@de-decms.com
kwessing3@linkedin.com
cstallion4@printfriendly.com
```

© 2019 Software Development Academy All Rights Reserved

Groups

1. Group extraction allows you to pick parts of the matched text
2. A group is delimited by parenthesis ( )
3. You can define multiple groups inside the pattern

© 2019 Software Development Academy All Rights Reserved

Example 12-regex/example-05.py

```
import re

lines = [
   "VII.   The Adventure of the Blue Carbuncle",
   "VIII.  The Adventure of the Speckled Band",
   "IX.    The Adventure of the Engineer's Thumb",
   "X.     The Adventure of the Noble Bachelor",
   "XI.    The Adventure of the Beryl Coronet",
]
pattern = re.compile("([IVX]+)\.\s+The Adventure of ([a-zA-Z' ]+)")
for l in lines:
    match = re.search(pattern, l)
    if match:
        print(match.groups())
```

```
$ python3 13-regex/example-05.py
```

```
('VII', 'the Blue Carbuncle')
('VIII', 'the Speckled Band')
('IX', "the Engineer's Thumb")
('X', 'the Noble Bachelor')
('XI', 'the Beryl Coronet')
```

© 2019 Software Development Academy All Rights Reserved

Exercise 4

- Write a pattern matching the following e-mails.
- Use groups to separate usernames and domains.
- Create a dictionary `{„user”: x, „domain”: }`, and print it

```
bjandourek0@stanford.edu
mtimothy1@deviantart.com
mdodding2@de-decms.com
kwessing3@linkedin.com
cstallion4@printfriendly.com
```

© 2019 Software Development Academy All Rights Reserved

Find All

1. Patterns can be used to search text for all occurencess of matching the pattern
2. To do so, use `re.findall` (which returns all matches) or `re.finditer` (which returns an iterator of all the matches).

© 2019 Software Development Academy All Rights Reserved

Example 12-regex/example-06.py

```
import re

lines = """
    VII.   The Adventure of the Blue Carbuncle
    VIII.  The Adventure of the Speckled Band
    IX.    The Adventure of the Engineer's Thumb
    X.     The Adventure of the Noble Bachelor
    XI.    The Adventure of the Beryl Coronet
"""
matches = re.findall("The Adventure of [a-zA-Z' ]+", lines)
print(matches)
it = re.finditer("The Adventure of [a-zA-Z' ]+", lines)
print(it)
```

```
$ python3 13-regex/example-06.py
```

```
['The Adventure of the Blue Carbuncle', 'The Adventure of the Speckled Band', "The Adventure of the Engineer's
Thumb", 'The Adventure of the Noble Bachelor', 'The Adventure of the Beryl Coronet']
<callable_iterator object at 0x0000000002ce0f60>
```

© 2019 Software Development Academy All Rights Reserved

Exercise 5

Using `re.findall` and provided file `holmes.txt`, find all the words which end the sentence. Count them.

© 2019 Software Development Academy All Rights Reserved