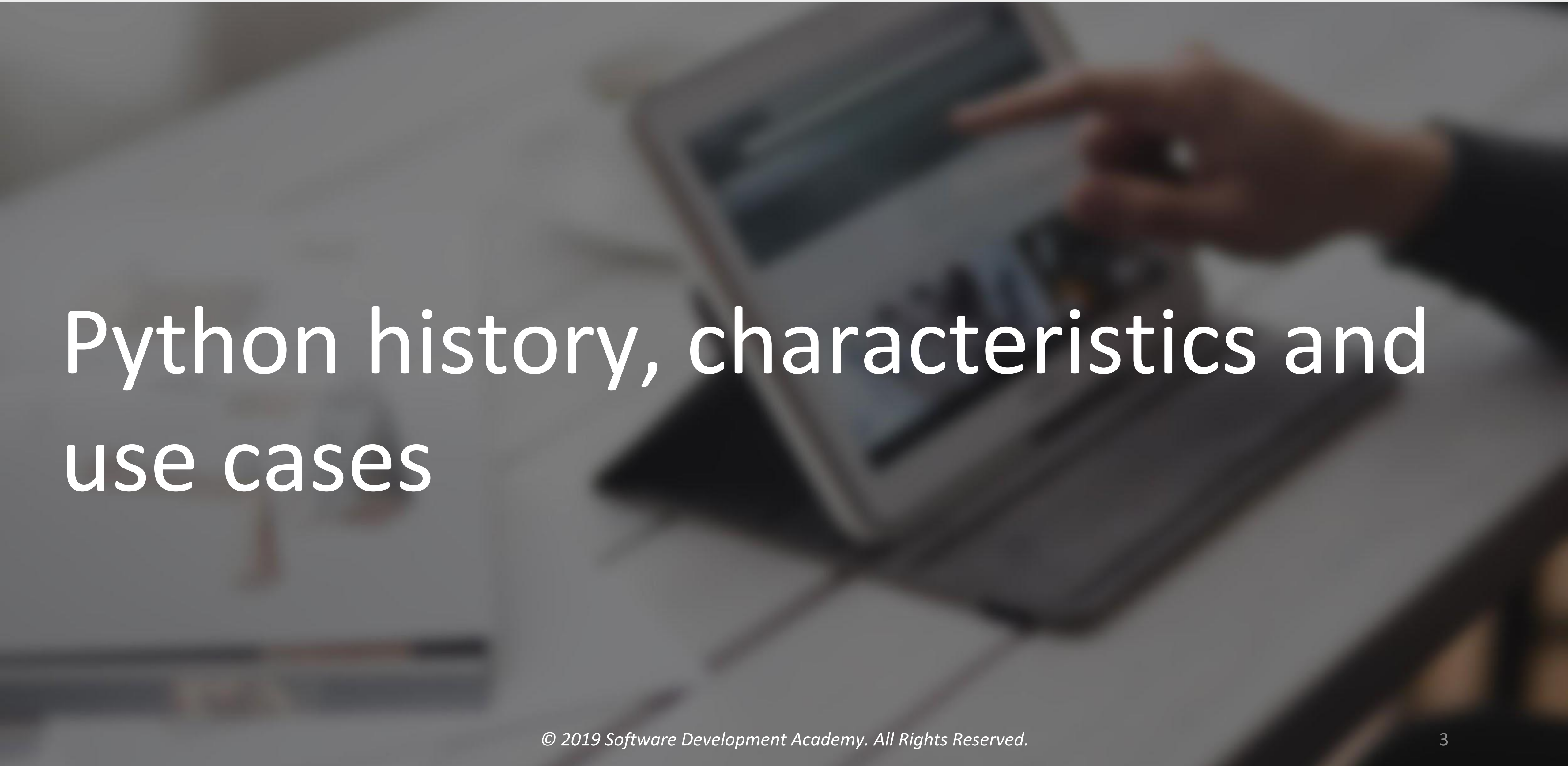
Python Fundamentals

Agenda

1. Python history, characteristics and use cases
2. Your first program: Hello, World!
3. Data types
4. Variables & Operators
5. UTF-8 compliance
6. String formatting and printing
7. Basic string operations
8. Collections
9. User input
10. Flow control
11. Introduction to functions
12. Object-oriented programming
13. File operations



Python history, characteristics and use cases

History of Python

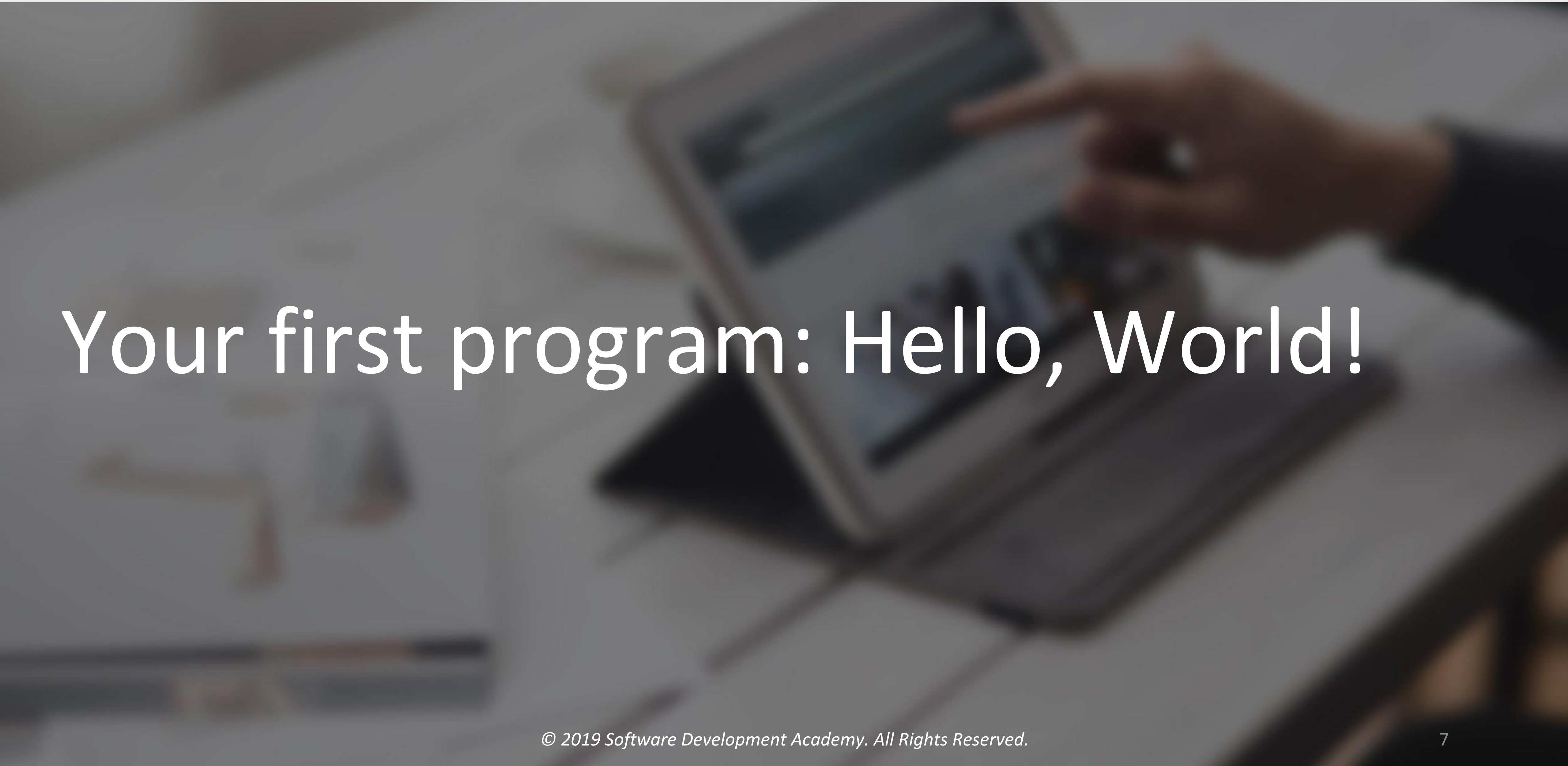
- Created by Guido van Rossum during the 1989 Christmas Holidays.
- Originally designed as a main scripting language for Amoeba operating system, but quickly outgrew its original purpose.
- First officially published in February 1991, reached its first major release (1.0.0) on January 26th, 1994.
- Python 3.0 has been released on December 8th, 2008
- Python 3.0 is the only officially supported version.

Python characteristics

- High-level
- Interpreted
- Dynamically typed
- Object-oriented
- Easily readable
- Whitespace sensitive
- Extensive standard library
- Free and open source

Common use cases of Python

- **Artificial Intelligence** and **Machine Learning** - SciKit, pyTorch, Tensor Flow.
- **Web backend** - Django, Pyramid, Flask, Bottle, paramiko.
- **Testing** - pytest, robot framework.
- **Task automatization** - Selenium, pyautogui, requests.
- **DevOps** - Ansible, Salt, OpenStack.
- **Data Science** - Pandas, IPython, Jupyter Notebook.



Your first program: Hello, World!

Prerequisites

Make sure that you have:

- Python 3.7 installed,
- PyCharm Community installed.

Instructions are located in module introduction document on Gitlab.

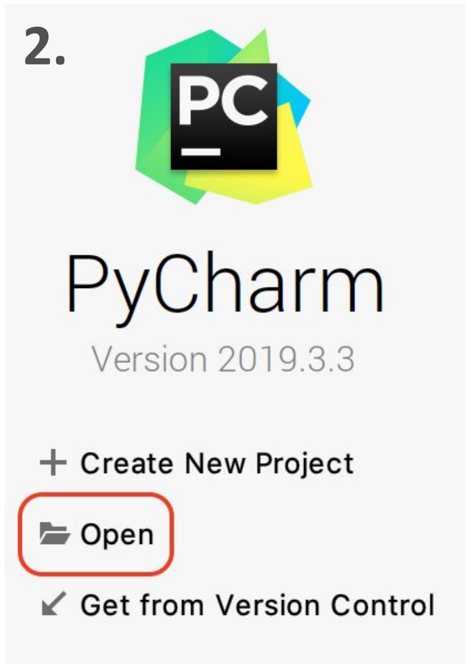
Your first program: Hello, World!

1. Create new folder which will contain files for Python Fundamentals module.
2. Open that folder in PyCharm by clicking **Open** and selecting it.
3. In PyCharm, inside that folder, create folder **01-hello-world**.
4. Inside that folder, create file **hello-world.py**.
5. Inside that file, type:

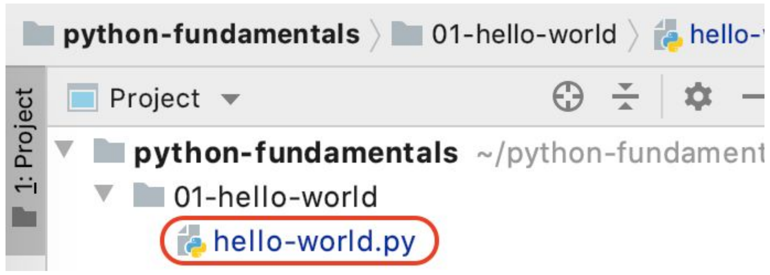
```
print("Hello, World!")
```

6. Execute the program in PyCharm by right clicking the file in the project files window and clicking **Run 'hello-world'**.
7. Notice the output produced by the program in PyCharm **console window** that appeared in the bottom area of PyCharm.

2.



6.



Note: You can also execute the program in Terminal / Command Line by changing active directory to **hello-world.py** location and executing command:

```
python3 hello-world.py
```

Hello world - explanation

In the hello world program, we have printed out a simple message in the console. We achieved that by calling a built-in `print()` **function**. This function accepts a **String parameter** inside brackets and prints it out in the console.

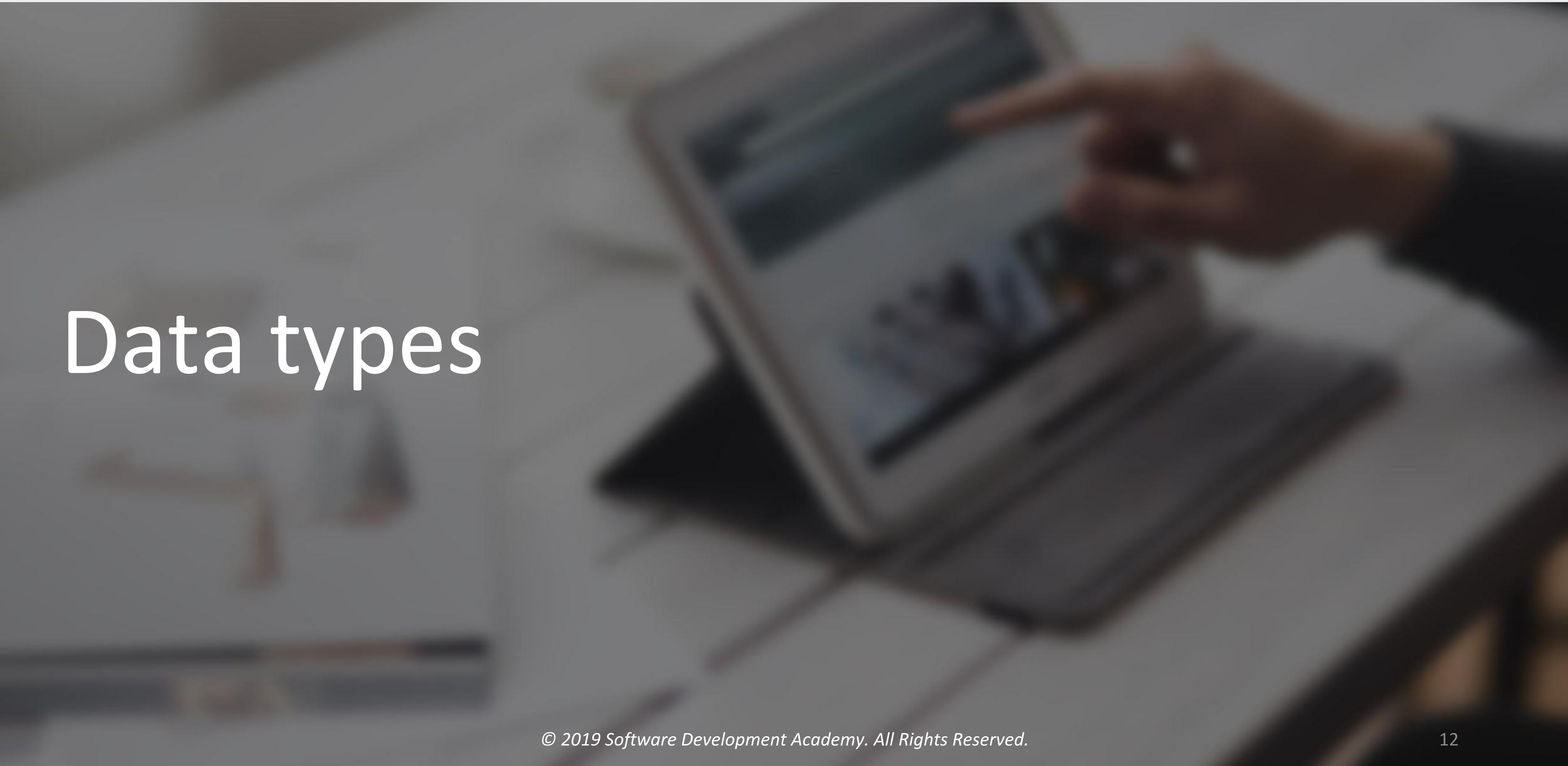
We will learn about functions later, for now think of them as **behaviours** - something gets done based on some input (**parameter**).

Function - explanation

A coffee machine preparing a coffee is a good example of a function.

Behaviour is always the same, a coffee is prepared for the user based on the coffee type choice (**parameter**), which can change anytime user interacts with the machine.

```
makeCoffee("Latte")
```



Data types

Data types

Python has data types that you should familiarise yourself with:

- **int** - used for integers,
- **float** - used for floating point numbers,
- **complex** - used for complex numbers,
- **str** and **bytes** - used for text sequences,
- **bool** - used for true/false values,
- **NoneType** - a special, “not defined” type, associated with None value.

Data types - examples

int	0, 1, -3, 128, 59007199254740991, 0b111, 0x7E3
float	0.123, 256.2, -3.14, 1e10
complex	(1-2j), (30+15j)
str	"house", "", "This is a sentence", 'single quotes'
bytes	b"This is a byte sequence", b'simple'
bool	True, False
NoneType	None

Printing different data types

We can print different data types exactly the same way how we printed the “Hello, World!” message in our first program.

```
# Printing different data types
print("A message.")
print(-17)
print(123.4)
print(False)
print(None)
```

Exercise time!

1. In PyCharm, create folder **02-data-types**.
2. Inside that folder, create file **data-types-exercise-01.py**.
3. Print out each data type in the console.

Checking data type

In Python, everything is an object. We will learn about **Object Oriented Programming** (**OOP**) later, for now think of an object as a box that contains something inside.

In Python, we are able to check of what particular data type an object is. To do so, we can use a built-in function `type()` that accepts the object as parameter.

```
type("What is my type?")
```

Checking data type

The `type()` function returns a text value (String) that describes data type of the function's parameter. To print it out in the console, we will pass returned result to the `print()` function as parameter.

```
# Checking data type and printing it
print(type("What is my type?"))
print(type(10.2))
print(type(False))
```

Exercise time!

1. In folder **02-data-types** create file **data-types-exercise-02.py**.
2. Print out type of each data type in the console.

Variables & Operators

Variables

Variables are used to **store data**. Think of variables as boxes that contain something (value). The stored value is what defines variable data type.

Python is a **dynamically typed language** so we don't have to define variable type when declaring it. Once we declare the variable by giving it a name, we also have to assign a value to it.

```
number = 10
```

Variable name declaration

Variable value assignment

Variables - rules

- You can change variable's value by assigning a new value to it.
- Variable name can only consist of **a-z**, **A-Z**, **0-9** and **_** symbols.
- Variable name can not start with a number.

Variables

Since **variable** stores some value, it also **represents the value**. We can use this fact to use variables as functions' parameters.

```python
# Declare and assign a variable.
number = 10
print(number)  # This prints out 10 in the console.
```

The **#** character instructs Python to ignore everything after it and is used for adding comments in code.

Exercise time!

1. In PyCharm, create folder **03-variables-and-operators**.
2. Inside that folder, create file **variables-exercise.py**.
3. Create variables for each data type.
4. Print out the variables in the console.
5. Assign new values of different types to existing variables.
6. Print out the variables in the console again.

Keywords

Keywords are names reserved by the language itself. It is forbidden to use them as variable, function or class names in your code. Doing so will result in SyntaxError being raised.

`and`	`as`	`assert`	`async`	`await`	`break`
`class`	`continue`	`def`	`del`	`elif`	`else`
`except`	`False`	`finally`	`for`	`from`	`global`
`if`	`import`	`in`	`is`	`lambda`	`None`
`nonlocal`	`not`	`or`	`pass`	`raise`	`return`
`True`	`try`	`while`	`with`	`yield`	

Operators

Arithmetic operators	+ - * ** / // %
Comparison operators	== != < > <= >=
Assignment operators	= += -= *= **= /= //= %= &= ^= |= <<= >>= |=
Identity operators	is, is not
Logical operators	and, or, not
Membership operators	in, not in
Bitwise operators	& AND, | OR, ^ XOR, ~ NOT, << left shift, >> right shift

Arithmetic operators

Arithmetic operators are used for arithmetic operations:

- **+** adds both numbers, can be also used to join multiple strings into one,
- **-** subtracts a number from another number,
- ***** multiplies both numbers, can be also used to repeat a string X times,
- **/** divides a number by another number,
- ****** raises a number to the power of another number,
- **//** calculates a root of a number power of another number,
- **%** calculates division remainder of both elements and is called **modulo operator**.

```python
# Arithmetic operators
print(1 + 2 + 5 - (2 * 2))
print(501.0 - 99.9999)
print(2 ** 3)
print(10.0 / 4.0)
print(10.0 // 4.0)
print(5 % 2)
```

```python
# Strings concatenation
name = "John"
greeting = "Hello, " + name
# Prints ‘Hello, John’
print(greeting)
```

```python
# String repeat
message = "Hey,ho"
# Prints 'Hey,hoHey,ho'
print(message * 2)
```

Assignment operators

Arithmetic operators are used for variable assignment operations:

- = assigns a value,
- **+=** adds a value to current variable value,
- **-=** subtracts a value from current variable value,
- ***=** multiplies current variable value by a value,
- **/=** divides current variable value by a value,
- ****=** raises current variable value to the power of a value,
- **//=** assigns a root of a value power of current variable value.

```python
# Assignment operators
num = 3
num += 2
print(num)
num -=1
print(num)
num *= 3
print(num)
num /= 2
print(num)
num **= 3
print(num)
num //= 2
print(num)
```

Comparison operators

Comparison operators are used for comparison operations:

- **==** returns true when both elements are equal,
- **!=** returns true when both elements are different,
- **<** returns true when an element is less than another element,
- **>** returns true when an element is greater than another element,
- **<=** returns true when an element is less or equal than another element,
- **>=** returns true when an element is greater or equal than another element.

```
# Comparison
john_1 = "John"
john_2 = "John"
print(john_1 == john_2)
print(1 != 1)
print(99 < 1.1)
print(99 > 1.1)
print(-32 <= -33)
print(123 >= 123)
```

Logical operators

Comparison operators are used for boolean algebra operations:

- **and** returns true when both statements are true,
- **or** returns true when either of the statements is true,
- **not** returns true when the statement is false, returns false otherwise.

```python
# Logical operators
print(True or False)
print(False and False and True)
print(not False)
is_greater = 40 > 30
print(not is_greater)
```

Membership operators

Membership operators are used for membership checking operations:

- **in** returns true when a value is present in a sequence (list, range, string, etc.),
- **not in** returns true when a value is not present in a sequence (list, range, string, etc.).

```python
# Membership operators
print("Membership operators")
print("fox" not in "cow, sheep, dog")
print("wine" in "France is famous for its wines")
```

Exercise time!

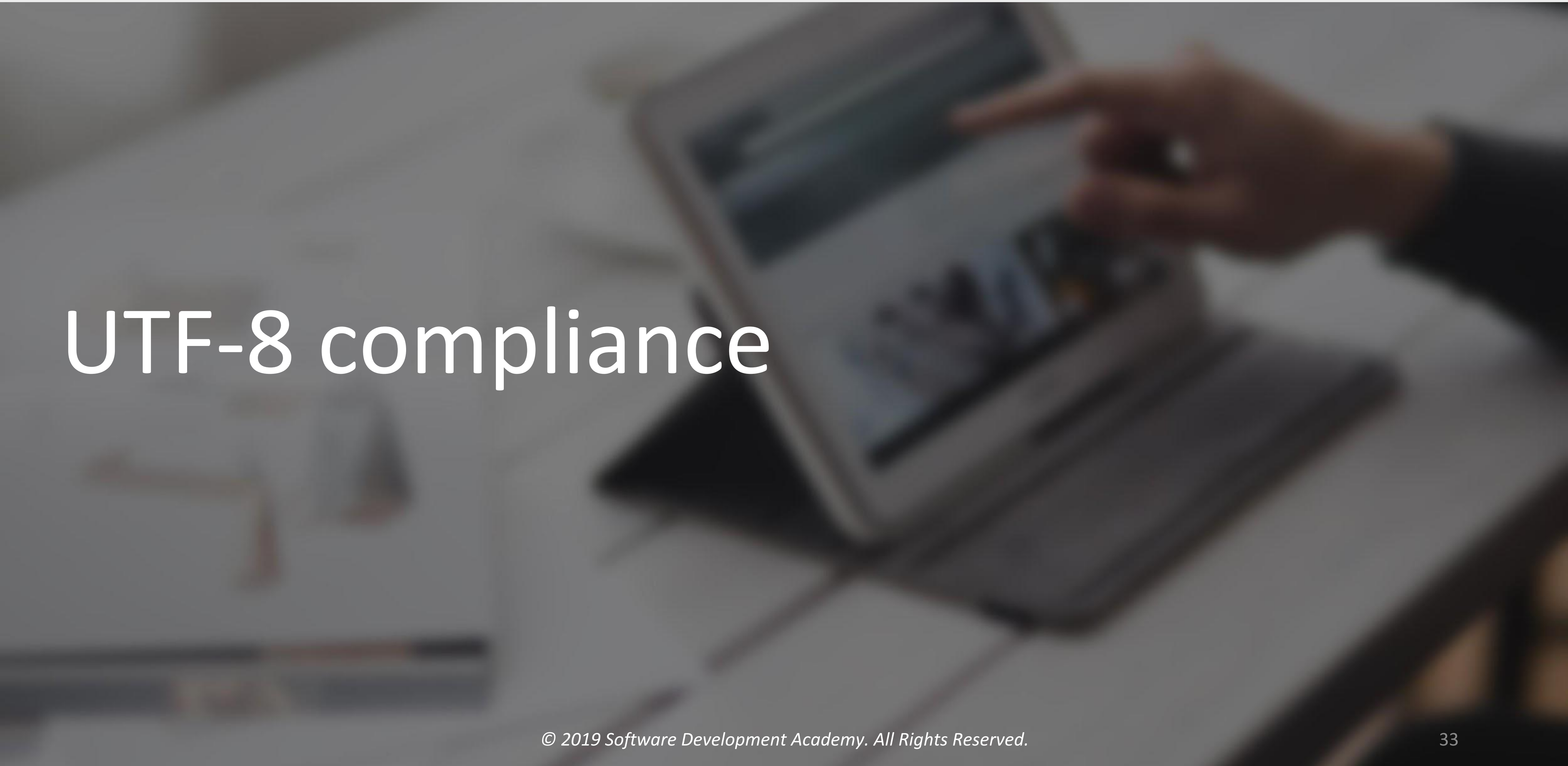
1. Try to determine what is printed on each line. Run file **operators-exercise.py** found on Gitlab to verify.

```
num = 10
print(num)
num += 1
print(num)
num -= 3
print(num)
num *= -0.5
print(num)
num **= 2
print(num)
num /= 3
print(num)
num //= 2
print(num)
num %= 2
print(num)
```

```
print(True = =  True)
print(1 ≠ 3)
print(2 < 3)
print(2 > 500)
print(2 ≥ 2)
print(2 ≤ 2)
```

```
print(5 + 5)
print(30.1 - 0.71)
print(2 * 2)
print(2 ** 3)
print(2.5 / 5.0)
print(10 % 3)
```

```
print(False or False or False)
print(False or True)
print(True and True)
print(True and False)
print(not False)
print(not True)
print(True and not False)
```



UTF-8 compliance

UTF-8

In 1963 the standard for information encoding **ASCII** was created. **ASCII** consisted originally of **128 characters**, including lowercase and uppercase letters, numbers and punctuation, each one encoded using **7 bits**.

Then came “**extended ASCII**” which used all **8 bits** to accomodate for more characters like á, é, ü and so on. It became apparent that neither 128 (7 bit) or 256 (8 bit) slots were enough to represent a very big number of characters consistently.

UTF-8

Unicode was created as a standard to represent characters from nearly all writing systems. It currently consists of more than 1,000,000 code points (they have the prefix “U+”).

UTF-8 is a method for encoding these code points and is the default encoding system for almost everything now.

UTF-8 compliance

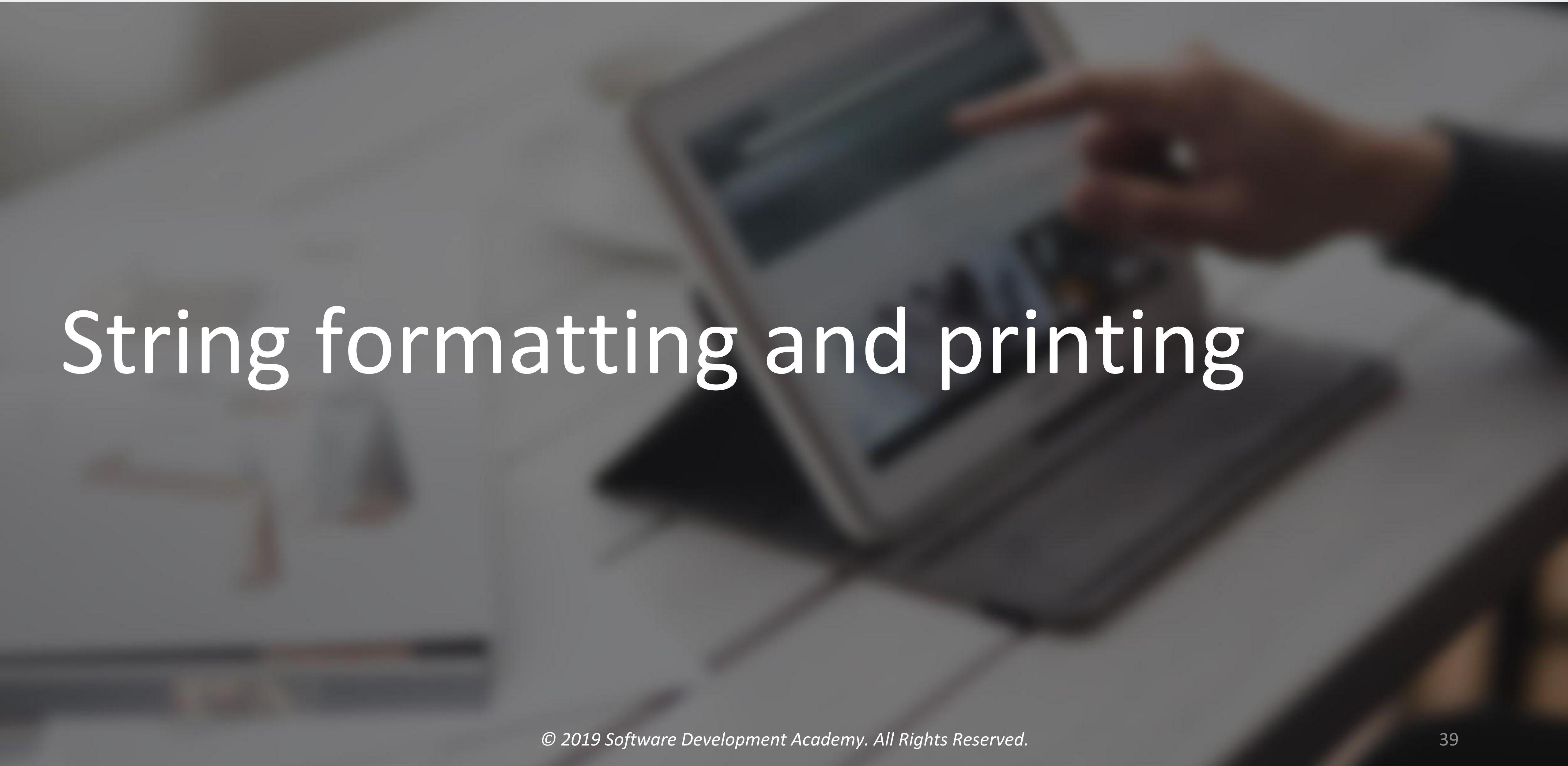
- Python 3.X is UTF-8 encoded by default.
- This means you can utilize diacritical marks in the string sequences.
- It is also possible to use them in variable, class and function names, but it is strongly discouraged.

UTF-8 compliance example

```python
# This is fine
pangram_en = "How vexingly quick daft zebras jump!"
pangram_fr = "Voix ambiguë d'un cœur qui, au zéphyr, préfère les jattes de kiwis."
pangram_de = '"Fix, Schwyz!", quäkt Jürgen blöd vom Paß.'
hello_cn = "你好，世界。"
print(pangram_en)
print(pangram_fr)
print(pangram_de)
print(hello_cn)

# This is STRONGLY discouraged
整数 = 7
print(整数)
```

Exercise time!

1. In PyCharm, create folder **04-utf-8-compliance**.
2. Inside that folder, create file **utf-8-exercise.py**.
3. Print out all characters of the Greek alphabet.



String formatting and printing

String printing

There are four main ways of printing data:

1. unformatted output,
2. printf-style formatting (old style),
3. str.format() formatting (new style),
4. string interpolation (formatted string literals or f-strings).

The print() function

We have already familiarised ourselves with the basic usage of the `print()` function. However, it is way more powerful that that. We can use it to:

- display multiple values at once,
- define a separator inserted between values,
- append a string after the last value.

print() - multiple values

We can provide from one to unlimited number of strings to be printed out. By default they will be separated by a space.

```
# Printing out multiple strings
print("What", "a", "lovely", "day", ".")
print("1", "2", 3, 4, 5)
fruit = "orange"
print("apple", "banana", fruit)
```

print() - separator

We can define a separator that will be inserted between the values by overwriting default value (space) of the function’s **sep** parameter.

```
# Printing out multiple strings with a separator
print("What", "a", "lovely", "day", ".", sep="-")
print("1", "2", 3, 4, 5, sep=" < ")
fruit = "orange"
print("apple", "banana", fruit, sep=" + ")
```

print() - ending

We can define an ending string that will be appended after the last value by overwriting default value ("\n" - new line) of the function's **end** parameter.

```python
# Printing out multiple strings with a separator and ending
print("What", "a", "lovely", "day", ".", sep="-", end="!\n")
print("1", "2", 3, 4, 5, sep=" < ", end=" <  ... \n")
fruit = "orange"
print("apple", "banana", fruit, sep=" + ", end=" = yummy\n")
```

Exercise time!

1. In PyCharm, create folder **05-string-formatting-and-printing**.
2. Inside that folder, create file **printing-exercise.py**.
3. Print out numbers from 0 to 5, each in new line, starting with “|START|” string and ending with “|END|” string using a maximum of 2 `print()` functions. The result should look like that:

```
|START|0|END|
|START|1|END|
|START|2|END|
|START|3|END|
|START|4|END|
|START|5|END|
```

Printf-style formatting

Strings in Python have a unique built-in operation that can be accessed with the **% operator**. It is commonly called “old-style” formatting. It makes a simple positional formatting very easy.

```
# Formatting and printing (old style)
name = "General"
last_name = "Kenobi"
print("Hello there, %s %s" % (name, last_name))
```

Printf-style formatting

It is also possible to refer to **variable substitutions by name** in the format string, if we pass a mapping to the **% operator**.

```
# Formatting and printing (old style)
name = "General"
last_name = "Kenobi"
print("Hello there, %(name)s %(last_name)s" % {"name": name,
"last_name": last_name})
```

Printf-style formatting

Variable substitutions by name make format strings easier to maintain and easier to modify in the future.

You do not have to worry about making sure the order you are passing the values in matches up with the order in which the values are referenced in the format string.

Of course, the downside is that this technique requires a little more typing.

str.format() formatting

Python 3 introduced a new way to do string formatting that was also later back-ported to **Python 2.7**.

This “new style” string formatting gets rid of the % operator special syntax and makes the syntax for string formatting more regular and intuitive to use.

str.format() formatting

Formatting is performed by calling `format()` function on a string object. We can use `format()` to do simple positional formatting, just like we could with “old style” formatting.

```
# Formatting and printing (new style)
name = "General"
last_name = "Kenobi"
print("Hello there, {} {}".format(name, last_name))
```

str.format() formatting

We can also refer to **variable substitutions by name** and use them in any order we want. This is quite a powerful feature as it allows for re-arranging the order of display without changing the arguments passed to the `format()` function.

```
# Formatting and printing (new style)
name = "General"
last_name = "Kenobi"
print("Hello there, {name} {last_name}".format(name=name,
last_name=last_name))
```

String interpolation

Python 3.6 added a new string formatting approach called **formatted string literals** or **f-strings**. This new way of formatting strings allows us to use embedded **Python expressions** inside string constants.

```
# Formatting and printing (string interpolation)
name = "General"
last_name = "Kenobi"
print(f"Hello there, {name} {last_name}")
```

String interpolation

String interpolation prefixes the string constant with the letter “f” - hence the name “**f-strings**”. This new formatting syntax is powerful, because we can embed arbitrary **Python expressions** like for example inline arithmetics.

```python
# Formatting and printing (string interpolation)
a = 2
b = 7
print(f"{a} times {b} raised to power of 2 is {(a * b) ** 2}.")
```

String interpolation

We can also define how a variable is displayed. For instance, we can define that a variable should be extended with spaces to exactly X characters if it is shorter.

```python
header1 = "Name"
header2 = "Age"
name = "John"
age = 22

print(f"| {header1:10} | {header2:10} |")
print("-" * 27)
print(f"| {name:10} | {age:10} |")
```

```
| Name       | Age        |
---------------------------
| John       |         22 |
```

String interpolation

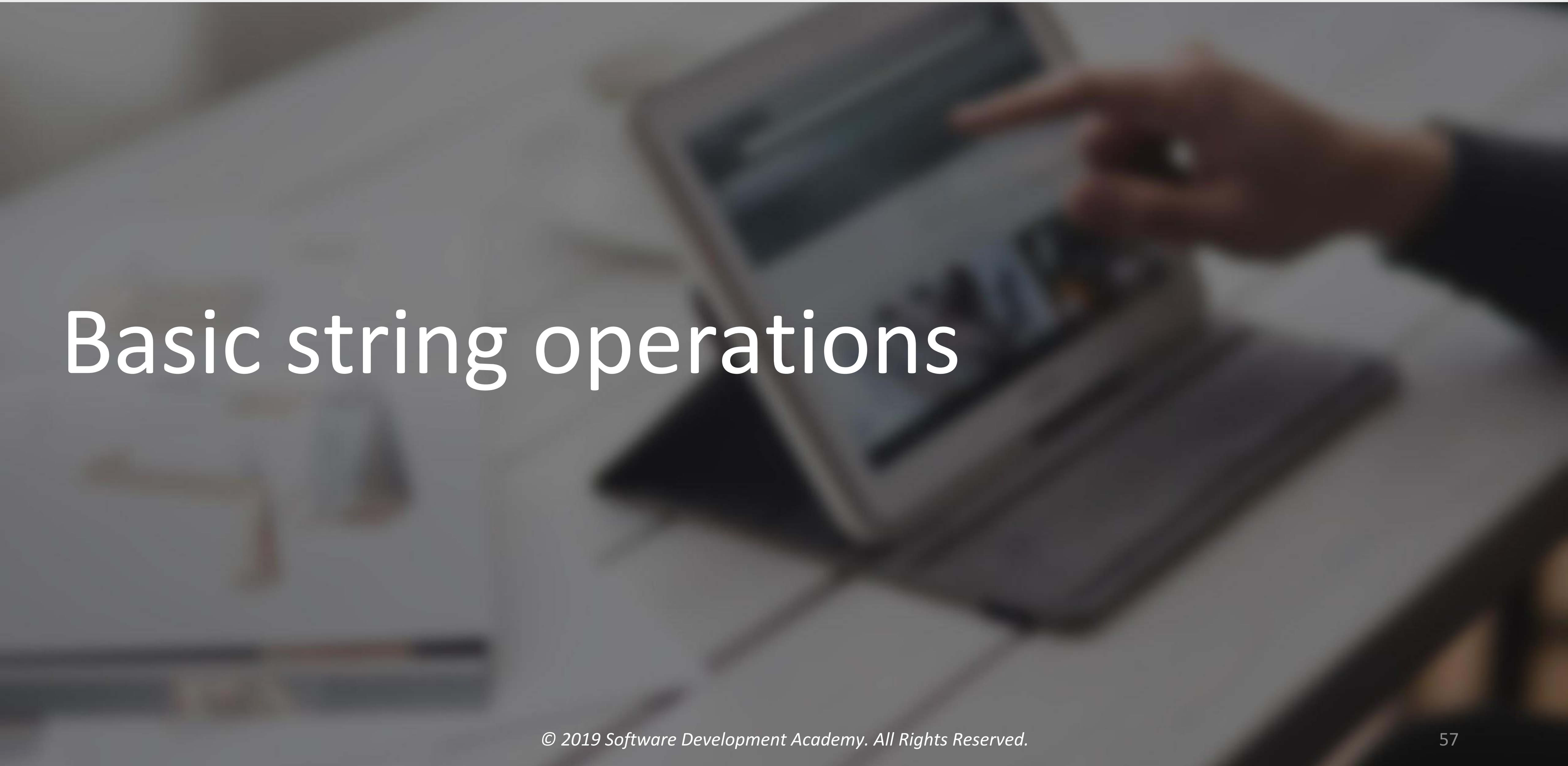
We can also define how many digits after comma should be displayed or display it as percentage.

```
# Changing how variable is displayed
n = 109.432188881111
print(f"{n:.3f}")  # prints out 109.432

voters_percentage = 0.71
print(f"{voters_percentage:.1%}")  # prints out 71.0%
```

Exercise time!

1. In folder **05-string-formatting-and-printing** create file **formatting-exercise-02.py**.
2. Print out a table with header row containing name, age and salary where each is accordingly extended to 15, 5 and 12 characters.
3. Print out a line to separate header from data columns that has precisely the table's length.
4. Print out 3 data rows in that table where all columns are even with header columns and salary has two digits after comma.

```
| Name            | Age   | Salary       |
------------------------------------------
| John Doe        |    27 |    123456.00 |
| John Wick       |    40 |     50000.00 |
| Jeff Bezos      |    45 | 999999999.95 |
```



Basic string operations

What strings are really?

A string is basically **a group of characters in a specified order**, called **a sequence of characters**. Sequences are **zero-indexed** what means that we count elements in the sequence starting with 0.

H	e	l	l	o	,		w	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11	12

And that is exactly how each string is stored in most programming languages, including Python.

len() function

The `len()` function returns a number that is equal to the number of characters in the string that.

```
# Print out amount of characters in the sentence
sentence = "Lorem ipsum dolor sit amet... "
print(len(sentence))  # prints out 29
```

.index() function

The `.index()` function returns a number that is equal to the position of first occurrence of a particular character in the string.

```
# Print out index of first 'o' character in the sentence
sentence = "Lorem ipsum dolor sit amet... "
print(sentence.index("o"))  # prints out 1
```

L	o	r	e	m		i	p	s	u	m		...
0	1	2	3	4	5	6	7	8	9	10	11	...

.count() function

The `.count()` function returns a number that is equal to the number of times a particular character occurs in the string.

```
# Print out amount of 'o' characters in the sentence
sentence = "Lorem ipsum dolor sit amet ... "
print(sentence.count("o"))  # prints out 3
```

Single character retrieval

The [ ] operator is used to retrieve a particular character of the string based on provided index.

```
# Print out 4th character of the sentence
sentence = "Lorem ipsum dolor sit amet ... "
print(sentence[3])  # prints out e
```

L	o	r	e	m		i	p	s	u	m		...
0	1	2	3	4	5	6	7	8	9	10	11	...

String slicing

The [ ] operator can be also used to retrieve a substring of of the string based on provided range containing of inclusive start index, colon and exclusive end index.

```
# Print out 'm ips' substring of the sentence
sentence = "Lorem ipsum dolor sit amet ... "
print(sentence[4:9])  # prints out 'm ips'
```

L	o	r	e	m		i	p	s	u	m		...
0	1	2	3	4	5	6	7	8	9	10	11	...

String slicing

If we only specify the start index of the [ ] operator, retrieved substring will start at that index and continue to the original string ending.

```
# Print out 'um dolor sit amet... ' substring of the sentence
sentence = "Lorem ipsum dolor sit amet... "
print(sentence[9:])  # prints out 'um dolor sit amet... '
```

L	o	r	e	m		i	p	s	u	m		...
0	1	2	3	4	5	6	7	8	9	10	11	...

String slicing

If we only specify the end index of the `[]` operator, retrieved substring will start at the original string beginning and end at that index.

```
# Print out 'Lorem ip' substring of the sentence
sentence = "Lorem ipsum dolor sit amet ... "
print(sentence[:8])  # prints out 'Lorem ip'
```

L	o	r	e	m		i	p	s	u	m		...
0	1	2	3	4	5	6	7	8	9	10	11	...

String slicing with skipping

The [] operator can be extended even further, to retrieve a substring skipping every nth character.

```
# Print out 'mis' substring of the sentence
sentence = "Lorem ipsum dolor sit amet ... "
print(sentence[4:9:2])  # prints out 'mis'
```

L	o	r	e	m		i	p	s	u	m		...
0	1	2	3	4	5	6	7	8	9	10	11	...

String reversing

The special syntax of the [] operator is used to reverse the string.

```
# Print out the sentence in reverse order
sentence = "Lorem ipsum dolor sit amet..."
print(sentence[::-1])  # prints out '...tema tis rolod muspi
meroL'
```

.upper() function

The `.upper()` function returns a string that is equal to the original string with characters raised to uppercase.

```
# Print out the sentence in uppercase
sentence = "Lorem ipsum dolor sit amet... "
print(sentence.upper())  # prints out the sentence in uppercase
```

.lower() function

The `.lower()` function returns a string that is equal to the original string with characters lowered to lowercase.

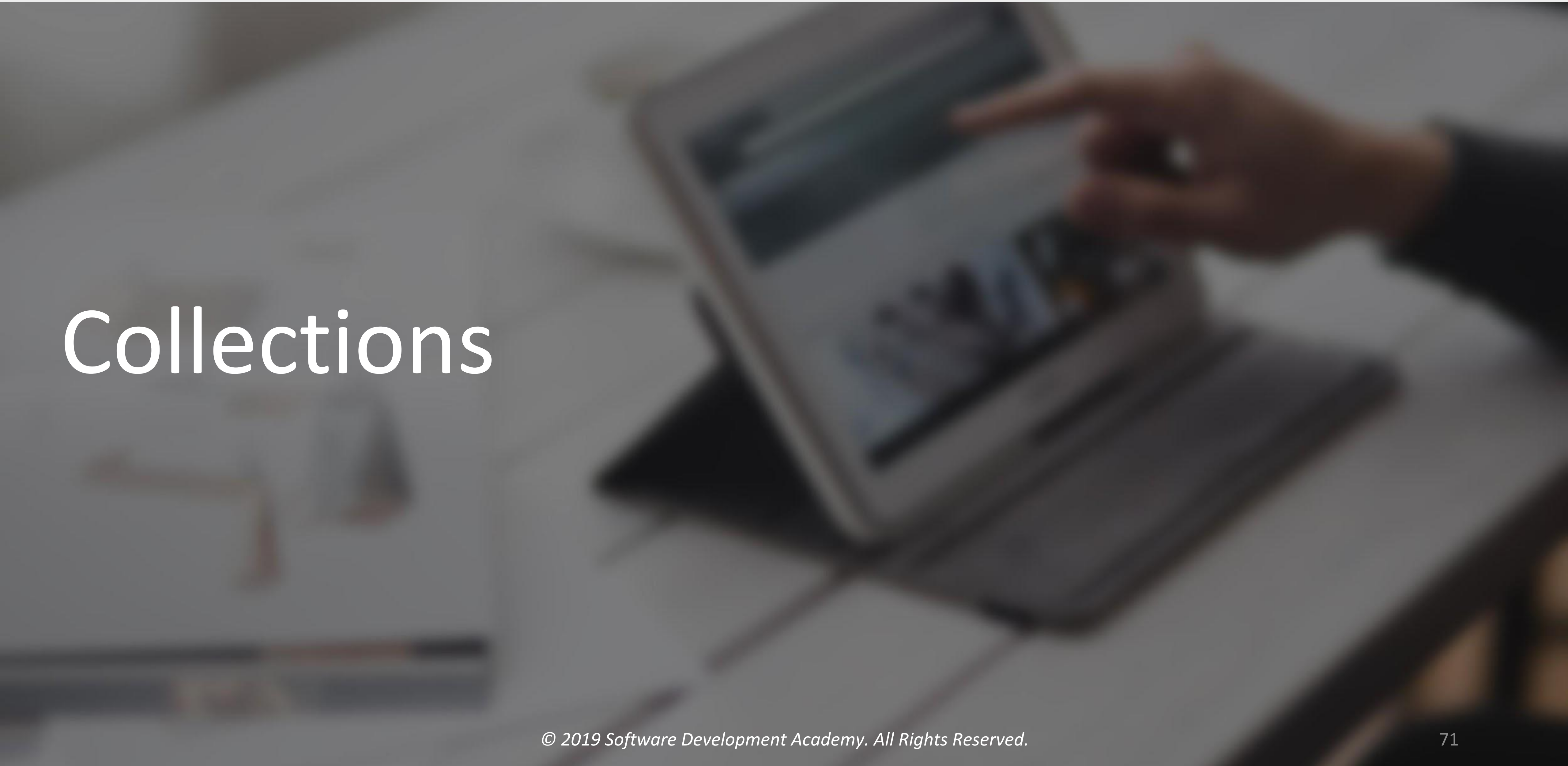
```
# Print out the sentence in lowercase
sentence = "Lorem ipsum dolor sit amet... "
print(sentence.lower())  # prints out the sentence in lowercase
```

Exercise time!

1. In PyCharm, create folder **06-basic-string-operations**.
2. Inside that folder, create file **strings-exercise.py**.
3. Write a program that operates on a provided string, that always has length of at least 10 characters.
    a. If the string **length is even**, retrieve a substring that is exactly **4** characters long and is exactly in the middle of the original string.
    b. If the string **length is odd**, retrieve a substring that is exactly **5** characters long and is exactly in the middle of the original string.

Tips: Make use of the `len()` and `int()` as well as the `%` and `[]` operators. Example of the `int()` function.

```
a = int(3 / 2)  # rounds down 1.5 to 1 and assigns it to a
```



Collections

What is a collection?

So far we have been working with single values. We have also learned, that string is really a sequence of characters. A collection is a container object which can hold zero or more objects of various types. Most important collections in Python are:

- List,
- Dictionary,
- Tuple,
- Set.

List

Lists can contain any type of variable and can contain as many variables as we wish. We must use the `[]` operator to initialize an empty list and the `len()` function to check how many elements are in the list.

```
# Declare and initialize a list variable
alphabet = []  # this is an empty list
print(f"Current length of 'alphabet': {len(alphabet)}")
```

List.append()

The `.append()` function is used to add an element to the list.

```
# Add some letters
alphabet.append("a")
alphabet.append("b")
alphabet.append("c")
print(f"Alphabet: {alphabet} (length: {len(alphabet)})")
```

List indexing

We have learned about zero-based indexing when we performed operations on strings. For lists it is exactly the same.

```python
# Indexing
print(f"The first letter of alphabet is '{alphabet[0]}'")
```

List.extend()

The `.extend()` function is used to add multiple elements to the list at once.

```
# Add some more letters
alphabet.extend(["f", "d", "g", "e"])
print(f"Alphabet (mixed): {alphabet} (length: {len(alphabet)})")
```

List.sort()

The `.sort()` function is used to sort elements in the list.

```python
# Sort the list
alphabet.sort()
print(f"Alphabet (sorted): {alphabet} (length: {len(alphabet)})")
```

Other list functions

To see all list functions we can use `help(list)` command, some of them are:

- `count(value)` - returns number of occurrences of value.
- `index(value)` - returns first index of the value.
- `insert(index, object)` - inserts the object before the index.
- `pop(index)` - removes and returns object at the index.
- `pop()` - removes and returns object at the last position.
- `remove(val)` - removes first occurence of the value.
- `clear()` - removes all items from the list.
- `reverse()` - reverses list order.

Exercise time!

1. In PyCharm, create folder **07-collections**.
2. Inside that folder, create file **list-exercise.py**.
3. Given list `[1, 2, 3, 4, 5]`:
   a. Use `len()` to print its length.
   b. Use `append()` to append a sixth element: another 2.
   c. Use `count()` to count the 2s in list.
   d. Use the same function to count 7s, which are not in the list. What's the result?
   e. Use `extend()` to extend it with: `[6, 7, 8]`.
   f. Use `index()` to check index of the 7. Use indexing to check you got the correct result.
   g. Use `insert()` to add a value 10 at index 0. Print the list to check what it did.
   h. Use `[-1]` indexing to check the value of the last element.
   i. Use `pop()` to check the value of the last element, removing it from the list.
   j. Use `remove()` to delete 4 from the list.
   k. Use `reverse()` and print the list..
   l. Use `sort()` and print it again.
   m. Use `clear()` to empty the list.

List slicing

1. Lists can be sliced.
2. Slicing operator is exactly the same as for string slicing:

```
[start_index:stop_index]
```

3. The result of slicing is a list.
4. Result list contains elements whose index in the original list fulfills the following condition: **`start_index ⩽ index < stop_index`**

List slicing

```
users = ["Alice", "Bob", "Chris", "Deborah"]

print(users)
print(users[0:3])  # prints out ['Alice', 'Bob', 'Chris']
print(users[1:2])  # prints out ['Bob']
print(users[:2])  # prints out ['Alice', 'Bob']
print(users[1:])  # prints out ['Bob', 'Chris', 'Deborah']
```

Exercise time!

1. In folder **07-collections** create file **list-slicing-exercise.py**.
2. Given list `users = ["User1", "UserChris", "User2", "Admin"]`:
    a. Print out a slice containing only “User2”.
    b. Print out a slice of all users, except the first one.
    c. Print out a slice of all users, except the “Admin”.
    d. Print out a slice of all users up to the 3rd one.

Dictionary

A dictionary is a data type similar to list, but works with **keys and values** instead of indexes.

Each value stored in a dictionary can be **accessed using a key**, which is any type of object (a string, a number, a list, etc.) instead of using its index to address it.

We must use the `{}` operator to initialize an empty dictionary.

Dictionary

```
# Declare and initialize an empty dictionary
phonebook = {}

# Add elements
phonebook["John"] = 111111111
phonebook["Jack"] = 222222222

print(phonebook)  # prints out {'John': 111111111, 'Jack':
222222222}
print(phonebook["Jack"])  # prints out 222222222
```

Dictionary

```
# Declare and initialize a dictionary
phonebook = {
    "John" : 111111111,
    "Jack" : 222222222
}

print(phonebook)  # prints out {'John': 111111111, 'Jack': 
222222222}
print(phonebook["Jack"])  # prints out 222222222
```

Dictionary element removal

The `.pop()` function and the **`del`** keyword deletes elements from the list.

```
phonebook = {"John": 111111111, "Jack": 222222222}

# Delete elements
del phonebook["John"]
phonebook.pop("Jack")
```

Dictionary.get()

The `.get()` function returns an object mapped to a key. We can also specify a default value when the key does not exist in the dictionary.

```
phonebook = {"John": 111111111, "Jack": 222222222}

# Find element by key that does not exist
print(phonebook.get("Dory"))  # prints out None
print(phonebook.get("Dory", 555555555))  # prints out 555555555
```

To see all dictionary functions we can use `help(dict)` command.

Exercise time!

1. In folder **07-collections** create file **dictionary-exercise.py**.
2. Given dictionary `{1: "one", 2: "two", 3: "three"}`:
   a. Use `len()` to print out its length.
   b. Using set-item operator `[]`, add a new key-pair: 4:"four".
   c. Using get-item operator `[]`, print out the value assigned to key 2.
   d. Using get-item operator `[]`, print out value for unassigned key, like 10. What happens?
   e. Using dictionary function `get(key)`, replace the previous get-item operator `[]` and print out the value for key 10. What happens?
   f. Using dictionary function `get(key, default)`, print out the value for key 10, this time setting default value to "unknown".
   g. Using dictionary function `get(key, default)`, print out the value for key 3. Set default value to "unknown".
   h. Use `pop()` to print out value assigned to 2. Print out the dictionary after using `pop()`.
   i. Create a new dictionary `{0: "zero"}`. Using `update()`, update main dictionary with values from the new dictionary. Print out the main dictionary.
   j. Using `clear()`, clear the dictionary.

Tuple

A tuple is a sequence of **immutable Python objects**. Tuples are sequences, just like lists. The differences between tuples and lists are, **the tuples cannot be changed** unlike lists and **tuples use parentheses**, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values.

Optionally we can put these comma-separated values between parentheses also.

Tuple

A tuple can be initialized in many ways.

```
# Declare and initialize a tuple
my_things = ("Dog", "Cat", 1997, 32.0, True)
my_things = "a", "b", "c", "d"
```

Tuple

We can access tuple elements and slice them just like lists.

```
# Access tuple element and slice the tuple
my_things = ("Dog", "Cat", 1997, 32.0, True)
print(my_things[0])
print(my_things[1:3])
```

We already know most of the tuple functions from lists and dictionaries. To see all dictionary functions we can use `help(tuple)` command.

Exercise time!

1. In folder **07-collections** create file **tuple-exercise.py**.
2. Given tuple `recipe = ("boil water", "insert egg", "wait 5min", "eat")`:
   a. Use `len()` to print out its length.
   b. Use get-item operator `[]`, to get 3rd step of the recipe.
   c. Print out a slice of the last two steps of the recipe.
   d. Count occurrences of “wait 5min” using `count()` function.
   e. Check whether “boil water” is the first step using `index()` function.

Set

A set is a collection which is **unordered and unindexed**. In Python sets are written with curly brackets.

We cannot access items in a set by referring to an index, since sets are unordered the items have no index.

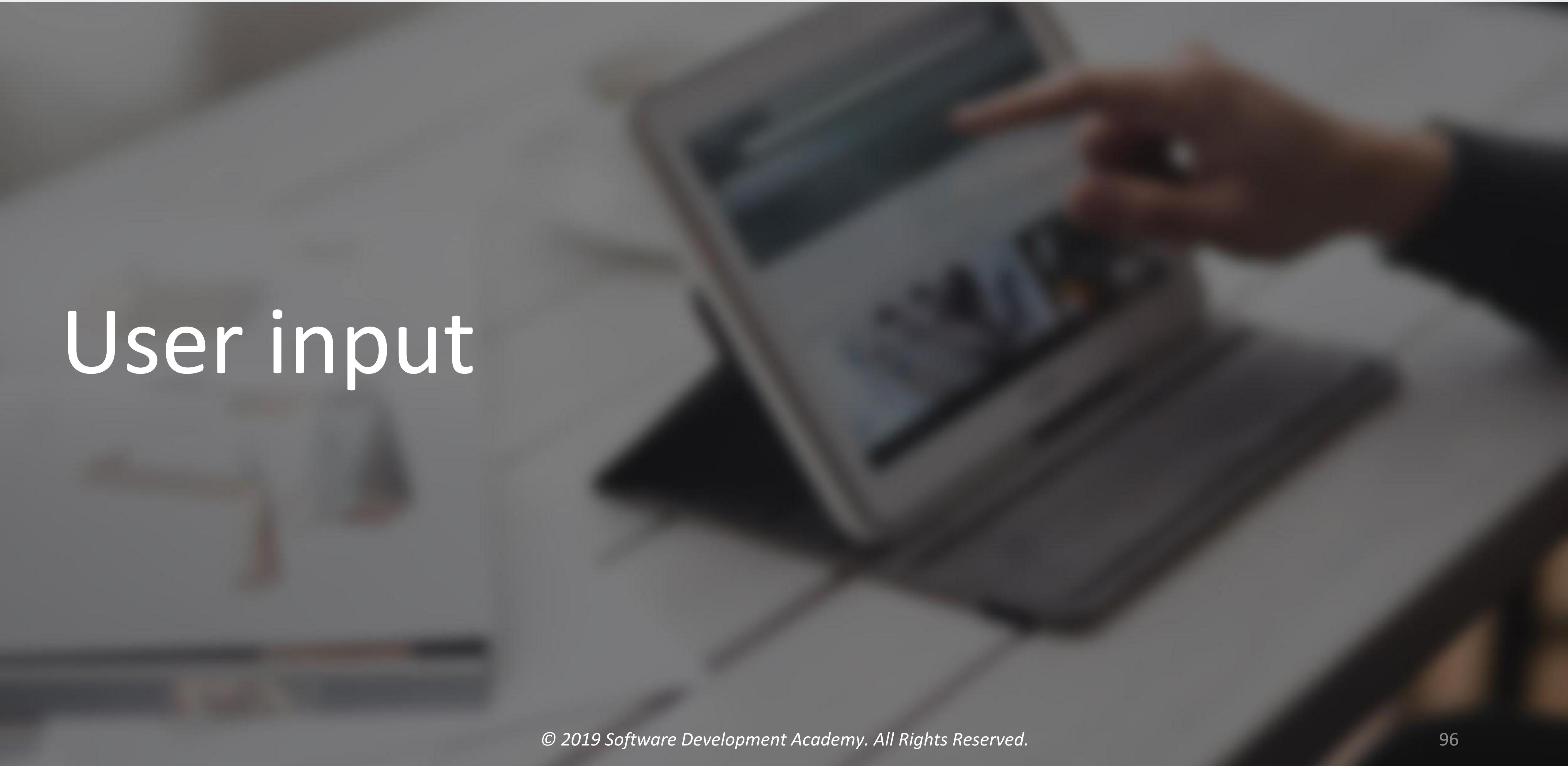
We already know most of the set functions from lists and dictionaries. To see all set functions we can use `help(set)` command.

Set

```python
# Declare and initialize a set
animals = {"Dog", "Cat", "Elephant"}

# Add an element
animals.add("Mouse")

# Add multiple elements
animals.update(["Bird", "Horse"])
```

Set

```python
# Declare and initialize a set
animals = {"Dog", "Cat", "Elephant"}

# Remove an element, throw an error if not present
animals.remove("Cat")

# Remove an element, DO NOT throw an error if not present
animals.discard("Cat")
```



User input

User input

All applications operate on data. One of the sources of data is user input.
Common ways of obtaining user input are:

- an interactive prompt,
- command line parameters.

Interactive prompt

The `input()` function is used to prompt user for data input. The program will halt the execution until an input is provided. The function's parameter is the message that should be displayed in the console when asking user for the input.

```
# Ask user for input and read it
print("Welcome to the interactive greeting system.")
user_name = input("Enter your name: ")
print(f"Hello, {user_name}!")
```

Exercise time!

1. In PyCharm, create folder **08-user-input**.
2. Inside that folder, create file **input-exercise-01.py**.
3. Write an application that prints a dictionary containing 3 country - capital pairs:
   a. Use input function to query user for a country and its capital three times for each pair (6 inputs total).
   b. Print out the resulting dictionary.
4. In folder **08-user-input** create file **input-exercise-02.py**.
5. Write the same application this time using `input()` only 3 times.
   a. User should input country and its capital in one input, separated by a comma "Japan,Tokyo".
   b. Use string `split(“,”)` function to split the string into a list of 2 substrings - the country and the city. The `split(“,”)` function splits a string by a programmer defined delimiter into a list of substrings, for example: *'Japan,Tokyo' → ['Japan', 'Tokyo']*
   c. Print out the resulting dictionary.

Command line parameters

Another way of providing input to our programs is to use **command line parameters**. Those parameters are provided when launching a program from the terminal / command after the program file name.

```
python3 my-program.py hello world!
```

Command line parameters

Before we are able to read those parameters in our program, we must **import a module** that will help us achieve that by adding simple line `import sys` at the top of our program file.

Once it is imported, we can use it to retrieve the command line parameters as a list of strings, where **the first item in the list is always full path to the application being executed and its name** and the next elements are the parameters we provided.

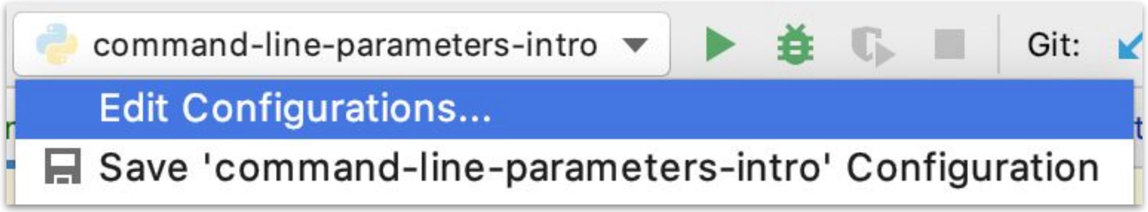
```
python3 my-program.py hello world!
```

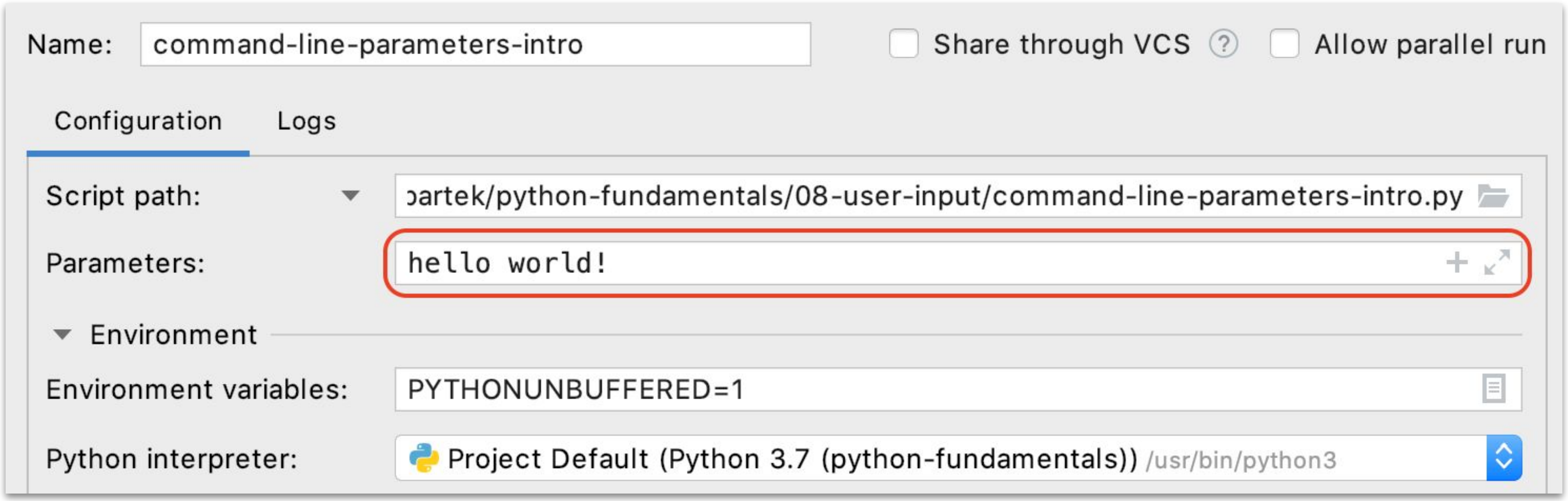
```python
# Import sys module and read command line parameters
import sys

# prints out full path to the application and its name
print(f"Application name: {sys.argv[0]}")
print(f"First argument: {sys.argv[1]}")  # prints out hello
print(f"Second argument: {sys.argv[2]}")  # prints out world!
```

Command line parameters

We can also provide command line parameters using PyCharm. To do so we must:

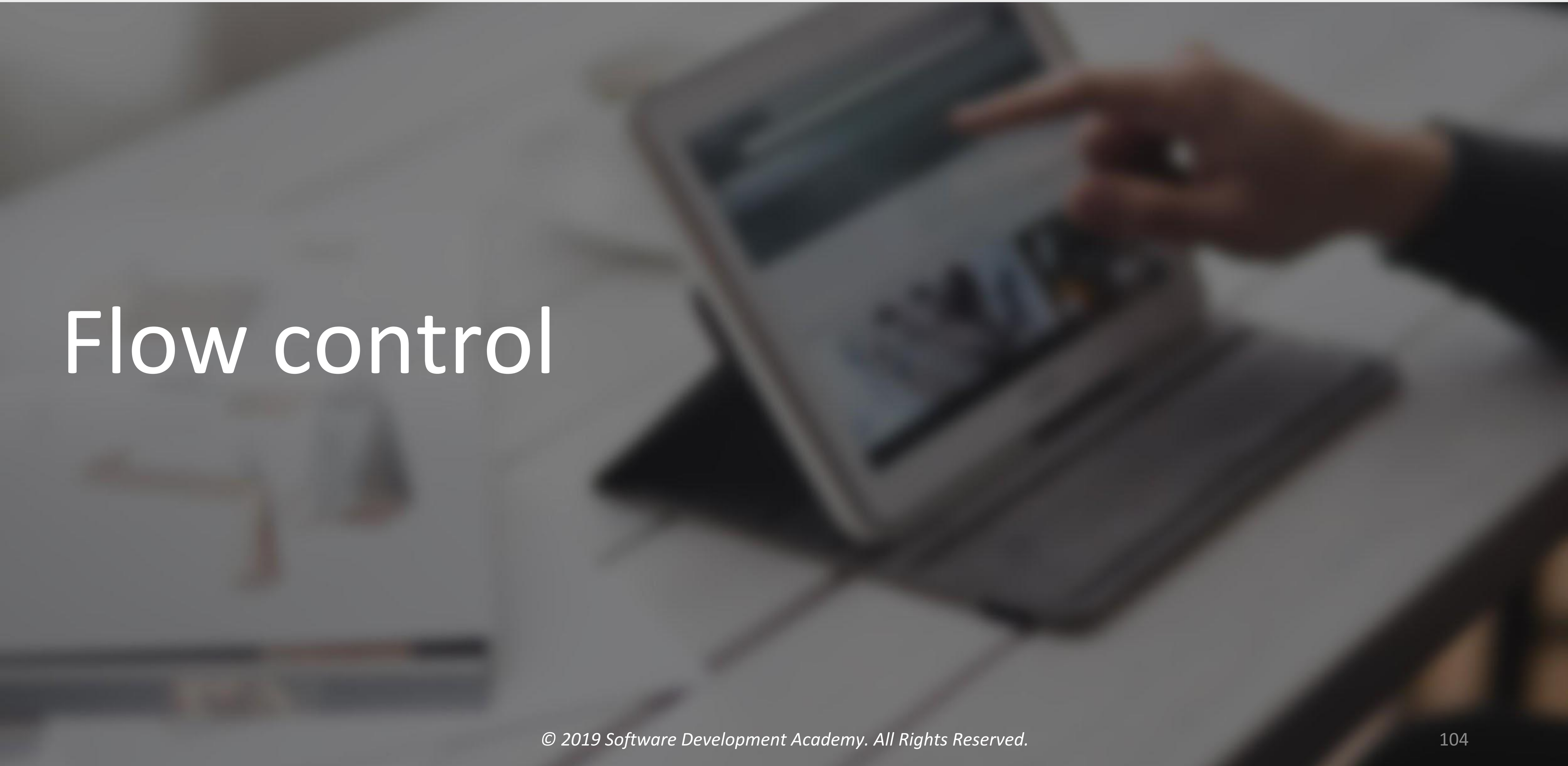
1. click ‘Edit Configurations…’ button in the upper right side of PyCharm,
2. provide the parameters in appropriate input field,
3. click ‘Apply’,
4. run the program.


command-line-parameters-intro
Git:
Edit Configurations...
Save 'command-line-parameters-intro' Configuration


Name: command-line-parameters-intro
Share through VCS
Allow parallel run
Configuration
Logs
Script path: artek/python-fundamentals/08-user-input/command-line-parameters-intro.py
Parameters: hello world!
Environment
Environment variables: PYTHONUNBUFFERED=1
Python interpreter: Project Default (Python 3.7 (python-fundamentals)) /usr/bin/python3

Exercise time!

1. In folder **08-user-input** create file **command-line-parameters-exercise-01.py**.
2. Write the same application as in the previous exercise (input-exercise-01.py), this time using command line parameters instead of `input()` function.
3. In folder **08-user-input** create file **command-line-parameters-exercise-02.py**.
4. Write the same application as in the previous exercise (input-exercise-02.py), this time using command line parameters instead of `input()` function.



Flow control

Flow control

Python supports statements that alter the top to bottom execution of the script. They are grouped into two groups: **conditional statements** (if, elif, else) and **loops** (while loop, for loop, break and continue statements).

Conditional statements

In the real world, we commonly must evaluate information around us and then choose one course of action or another based on what we observe.

If the weather is nice, **then** I'll mow the lawn.

In a Python program, the **if statement** is how you perform this sort of decision-making. **It allows for conditional execution of a statement or group of statements based on the value of an expression.**

The if statement

The most basic form of the **if statement** in its simplest form.

```
if <expr>:
    <statement>
```

`<expr>` is an expression evaluated in Boolean context, `<statement>` is a valid Python statement, which must be indented. If `<expr>` is true (evaluates to a value that is True), then `<statement>` is executed. If `<expr>` is false, then `<statement>` is skipped over and not executed.

The if statement

```python
x = 0
y = 3

if x > y:  # evaluates to false, message is not displayed
   print(f"{x} is greater than {y}")

if x < y:  # evaluates to true, message is displayed
   print(f"{x} is lesser than {y}")
```

Indentation

Python is all about the indentation. To execute more than one statement in an if block, all statements must be indented accordingly.

Indentation is used to define compound statements or blocks. In a Python program, contiguous statements that are indented to the same level are considered to be part of the same block.

```
if <expr>:
    <statement>
    <statement>
    ...
    <statement>
<following_statement>
```

The else clause

We already know how to use an if statement to conditionally execute a single statement or a block of several statements. But what if we want to evaluate a condition and take **one path if it is true** but specify an **alternative path if it is false**. This is accomplished with an **else clause**.

```
if <expr>:
    <statement(s)>
else:
    <statement(s)>
```

The else clause

```
x = 0
y = 3

if x > y:  # evaluates to false, message is not displayed
   print(f"{x} is greater than {y}")

else:  # is executed when if statement evaluates to false,
message is displayed
   print(f"{x} is less than {y}")
```

The elif clause

There is also syntax for branching execution based on several alternatives. For this, use **one or more elif** (short for else if) clauses.

Python evaluates each `<expr>` in turn and **executes the suite corresponding to the first that is true**. If none of the expressions are true, and an else clause is specified, then its suite is executed.

```
if <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
     ...
else:
    <statement(s)>
```

The elif clause

An arbitrary number of elif clauses can be specified. The **else clause is optional**. If it is present, there can be only one, and it must be specified last.

At most, one of the code blocks specified will be executed. If an else clause isn't included, and all the conditions are false, then none of the blocks will be executed.

```
if <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
    ...
else:
    <statement(s)>
```

The elif clause

```
x = 0
y = 3

if x > y:  # evaluates to false, message is not displayed
   print(f"{x} is greater than {y}")
elif x == 3:  # evaluates to false, message is not displayed
   print(f"{x} is equal to {y}")
else:  # is executed when none of the if/elif statement
evaluates to true, message is displayed
   print(f"{x} is less than {y}")
```

Exercise time!

1. In PyCharm, create folder **09-flow-control**.
2. Inside that folder, create file **if-statement-exercise.py**.
3. Write an application that:
   a. Asks user for a number from 1 to 7.
   b. If the number provided by user is smaller than 1, prints out “There are no negative number days!”.
   c. For input number 1, prints out “You chose Monday”.
   d. If the number provided by user is greater than 7, prints out “There are only 7 days in a week!”.

Loops

To iterate means to execute the same block of code over and over. A programming structure that implements iteration is called a **loop**.

With **indefinite iteration**, the number of times the loop is executed isn't specified explicitly in advance. Rather, the designated block is executed repeatedly as long as some condition is met.

With **definite iteration**, the number of times the designated block will be executed is specified explicitly at the time the loop starts.

The while loop

When a while loop is encountered, `<expr>` is first evaluated in Boolean context. If it is true, the loop body is executed. Then `<expr>` is checked again, and if still true, the body is executed again.

This continues until `<expr>` becomes false, at which point program execution proceeds to the first statement beyond the loop body.

```
while <expr>:
    <statement(s)>
```

The while loop

This program prints out 1, 2, 3, 4, 5 each in new line.

```
# Execute while body block as long as n < 5
n = 0
while n < 5:
    n += 1  # increment n with each loop execution
    print(n)
```

Loop termination

Python provides two keywords that terminate a loop iteration prematurely.

The **`break`** statement immediately **terminates a loop entirely**. Program execution proceeds to the first statement following the loop body.

The **`continue`** statement immediately **terminates the current loop iteration**. Execution jumps to the top of the loop, and the controlling expression is re-evaluated to determine whether the loop will execute again or terminate.

The while loop

This program will print out 2 and 3 each in new line.

```python
# Execute while body block as long as n < 5
n = 0
while n < 5:
    n += 1  # increment n with each loop execution
    if n == 4:  # if n is 4 then exit the loop
        break
    if n == 1:  # if n is 1 then start next iteration
        continue
    print(n)
```

The else statement in while loop

Python allows an optional **else** clause at the end of a **while** loop. When <additional_statement(s)> are placed in an **else** clause, they will be executed only if the loop terminates “by exhaustion”. That is, if the loop iterates until the controlling condition becomes false. If the loop is exited by a **break** statement, the **else** clause won’t be executed.

```
while <expr>:
    <statement(s)>
else:
    <additional_statement(s)>
```

The else statement in while loop

```
# Execute while body block as long as n < 5
n = 0
while n < 5:
   n += 1  # increment n with each loop execution
   print(n)
else:
   print("Done.")  # print out Done. only when while loop
condition is exhausted
```

Exercise time!

1. In folder **09-flow-control** create file **while-loop-exercise.py**.
2. Write an application that:
   a. Asks user for an input in a loop and prints it out.
   b. If the input is equal to “exit”, program terminates printing out provided input and “Done.”.
   c. If the input is equal to “exit-no-print”, program terminates without printing out anything.
   d. If the input is equal to “no-print”, program moves to next loop iteration without printing anything.
   e. If the input is different than “exit”, “exit-no-print” and “no-print”, program repeats.

The for loop

`<iterable>` is a collection of objects - a list for instance. The `<statement(s)>` in the loop body are denoted by indentation, as are all Python control structures.

The loop body is **executed once for each item** in `<iterable>`. The loop variable `<var>` takes on the value of the next element in `<iterable>`.

```
for <var> in <iterable>:
    <statement(s)>
```

The for loop

Printing out all elements of a collection is really that simple.

```
animals = ["Dog", "Cat", "Fish"]

# Print out all animals in the list
for animal in animals:
   print(animal)  # prints out a single animal
```

The for loop

The **`break`** and **`continue`** statements as well as **`else`** statement are fully supported in the **`for`** loop just like they are in the **`while`** loop.

The range() function

Very often, we would like to execute a loop given amount of times. We could create a simple list of numbers and iterate over it.

```
    for i in (1, 2, 3).
```

But what if we wanted to do it way more times?

That is where the `range()` function comes in handy.

The range() function

The `range()` function returns an iterable that contains integers starting with `start`, up to but not including `end`.

If specified, `step` indicates an amount to skip between values - just like the step value used for string and list slicing.

```
    range(start, stop, step)
```

The range() function

```
# Print 0, 1, 2
for i in range(3):
    print(i)

# Print -3, -2, -1, 0
for i in range(-3, 1):
    print(i)
```

```
# Print 0, 3, 5, 7, 9
for i in range(3, 11, 2):
    print(i)

# Print -1, -2, -3
for i in range(-1, -4, -1):
    print(i)
```

Exercise time!

1. In folder **09-flow-control** create file **for-loop-exercise.py**.
2. Write an application that prints a sum of all even numbers between 2020 and 3030.

Introduction to functions

What are functions?

Functions are a convenient way to divide our code into useful blocks, allowing us to order our code, make it more readable, reuse it and save some time.

Functions are used to extract pieces of code that should be reused, potentially with different arguments thus generating different outcome.

Functions must be declared before piece of code that is calling them.

Functions

Functions in python are defined using the **def** keyword, followed with the function’s name and zero or more parameters in parentheses.

```
# Define function print_hello_world()
def print_hello_world():
   print("Hello world from function!")

# Call function print_hello_world()
print_hello_world()
```

Functions

Function parameters should be placed inside parentheses, so that the function can operate on them.

```python
# Define function greet_by_name(name)
def greet_by_name(name):
   print(f"Hello, {name}")

# Call function greet_by_name(name) passing "John" as name
greet_by_name("John")
```

Function parameters

Function parameters can be:

- mandatory,
- optional (keyword parameters).

Arguments to mandatory parameters are usually passed without naming them.
Arguments to optional parameters are usually named in function call.

Optional parameters must always be defined after mandatory parameters.

Function parameters.

Parameters in functions can be assigned default values, which can be overwritten by passing values in function call statement.

```
# Define function greet_by_name(name) with default argument value
def greet_by_name(name="World!"):
    print(f"Hello, {name}")


# Call function greet_by_name(name) using default value of the argument
greet_by_name()  # prints 'Hello, World!'
# Call function greet_by_name(name) passing "John" as name
greet_by_name("John")  # prints 'Hello, John'
greet_by_name(name="John")  # prints 'Hello, John'
```

Functions

Functions in python can return a value using the **return** keyword. Functions that don't explicitly use the **return** keyword also return value equal to **None**.

```python
# Define function calculating volume of the cube
def calculate_volume_of_the_cube(wall_length):
    return wall_length ** 3

volume = calculate_volume_of_the_cube(6)
print(volume)  # prints 216
```

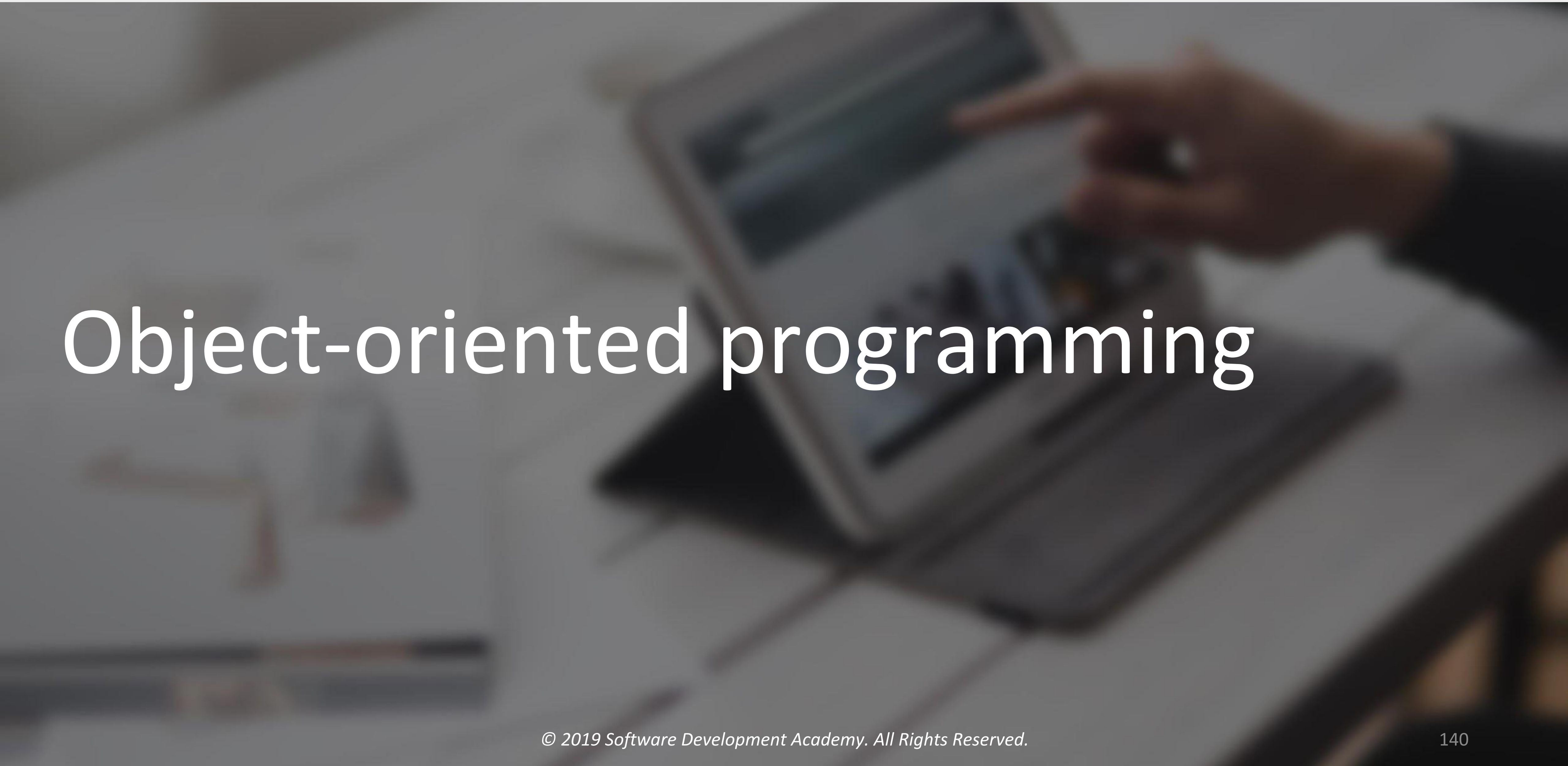
Exercise time!

1. In PyCharm, create folder **10-functions**.
2. Inside that folder, create file **functions-exercise-01.py**.
3. Write a function `max_of_three(a, b, c)` which returns the biggest of the three numbers.

Exercise time!

1. In folder **10-functions**, create file **functions-exercise-02.py**.
2. Write a function `print_rectangle` which prints out a rectangle made of character and wall size defined by user.
3. Make the character parameter optional with default value ‘#’.
4. Make the wall size parameter mandatory.
5. For wall_size = 3 the output should be:

```
###
###
###
```



Object-oriented programming

Object-oriented paradigm

Object-oriented paradigm is based on the concept of objects. Those objects contain fields representing their **state** (**variables**) and **methods** (**object-specific functions**) that are able to read and modify the state.

Object-oriented programming is based on the idea of defining such objects and their interactions to simplify complex problems, separate responsibilities, parallelize code execution and more.

Class and object

Class is essentially a template defining the object by specifying fields (variables), methods (functions) and default state. Think of the class as of a definition or a blueprint for creating an object.

Object is a result of creating an instance of a class. There can be infinite number of objects of a particular class. Objects get their variables and functions from classes.

Class

Class is defined using the **class** keyword followed by desired class name. Then with appropriate indentation, class variables can be defined. The `__init__(self)` function is called when an object of a class is being instantiated (created) and then initialized. The `self` parameter is required in every class' function so that it can refer to itself.

```
class Animal:
    name = ""  # class variable
    age = 0  # class variable

    def __init__(self):  # special method used for instantiation - that is
object creation
        self.name = "Jenna"  # setting default name when creating the object
        self.age = 2  # setting default age when creating the object

    def print_details(self):  # class method printing state of the instance
        print(f"Name: {self.name}, age: {self.age}")
```

Object

Once the class is defined, its instances (objects) can be instantiated (created), initialized and assigned to a variable. This is done by calling the `__init__(self)` function (`my_dog = Animal()`). Once the object of the class is created, we can use it just like we used other objects (for example string) before.

To access object's variables or call functions, we must refer to their names `my_dog.age = 3` or `my_dog.print_details()`.

```
class Animal:
    ...

my_dog = Animal()
my_dog.print_details()  # call function on particular object (my_dog)
print(my_dog.name)  # access particular object's field variable (my_dog)
my_dog.age = 3  # change particular object's field value (my_dog)
```

Object

Each of the instances (objects) of a class has its own state. Changing the state in one object does not change it in other.

```
class Animal:
    ...

my_puppy = Animal()  # create my_puppy instance of Animal
my_older_dog = Animal()  # create my_older_dog instance of Animal
my_puppy.age = 1
my_puppy.name = "Rex Junior"
my_older_dog.age = 10
my_older_dog.name = "Rex Senior"

# prints out 'My puppy: Rex Junior, 1 and my older dog: Rex Senior, 10'
print(f"My puppy: {my_puppy.name}, {my_puppy.age} and my older dog: 
{my_older_dog.name}, {my_older_dog.age}")
```

The __init__() function

Setting a default object’s state is not always desired. When we were creating our animals, we had to create the object and set the name and age. However, we can extend the `__init__(self)` function with optional parameters. This way we have default values that can be overwritten!

```
def __init__(self, name="Jenna", age=2):
   self.name = name  # setting name when creating the object
   self.age = age  # setting age when creating the object
```

Exercise time!

1. In PyCharm, create folder **11-oop**.
2. Inside that folder, create file **oop-exercise-01.py**.
3. Create Vehicle class with fields: **name**, **type**, **color** and **value** and methods: **description()** that returns a string describing the vehicle.
4. The Vehicle class initialization method should require **name** and **price** parameters and have default values for **type** and **color**.
5. Create vehicles list.
6. Write a loop that asks user 3 times for input regarding vehicle creation. With each loop iteration user should provide name and price and should be asked if he wants to provide type and color. If the user responds yes, he should be asked for the type and color. Once the input is collected, create new vehicle based on the input and add it to the vehicles list.
7. In a loop, print details of each car in the cars list.

Public vs. private

In our Animal class, we were able to access the age field and set any value we wanted.

Imagine, that other people will use our Animal class, and some of them will try to put in negative number in that property. This should not be possible.

Private variable

In Python world, attributes prefixed with a single underscore character are treated as internal class variables that should not be touched by people using particular class. However, this is only a convention and the field is still accessible.

```
class Animal:
   _name = ""  # private class variable, still accessible
   _age = 0  # private class variable, still accessible
    …


my_dog = Animal()
print(my_dog._name)  # prints out the variable without issues
print(my_dog._age)  # prints out the variable without issues
```

Mangled variable

To introduce a real private field that is not directly accessible, we must use double underscore. Such variable is called a mangled variable.

```
class Animal:
    __name = ""  # mangled class variable, inaccessible
    __age = 0  # mangled class variable, inaccessible
     ...

my_dog = Animal()
print(my_dog.__name)  # throws and error
print(my_dog.__age)  # throws an error
```

Mangled variable

Once the mangled variable is introduced, we are certain that nobody will change the animal’s age to a negative number, simply because they can’t change anything now.

This is quite troubling, because we want to allow users to change the age of their animals to correct values.

To remedy the problem we can create methods in Animal class that will access the mangled variables and return them for printing purposes.

Mangled variable

```
class Animal:
    __name = ""  # mangled class variable, inaccessible
    __age = 0  # mangled class variable, inaccessible

    ...

    def set_age(self, age):
        if age > 0:
            self.__age = age
        else:
            print("Age must be greater than 0.")

    def get_age(self):
        return self.__age


my_dog = Animal()
my_dog.set_age(3)  # sets the age
print(my_dog.get_age())  # gets the age
```

Properties

Calling functions to set or get a variable value is not really the Pythonic way of dealing with things. That is where the **properties** come with the rescue.

Properties are class’ special attributes and are used to truly encapsulate class’ fields.

A property can have a **getter**, **setter** and **deleter** method. Property’s methods share its name and are distinguished by the operator.

Properties

```
class Animal:

    ...

    @property  # getter
    def age(self):
        return self.__age

    @age.setter  # setter
    def age(self, age):
        if age > 0:
            self.__age = age
        else:
            print("Age must be greater than 0.")

    @age.deleter  # deleter
    def age(self):
        del self.__age
```

```
my_dog = Animal()
my_dog.age = 3  # sets the age
print(my_dog.age)  # gets the age
```

Property method names must be the same for all three actions (get, set and delete) and the **method name becomes the property name** when accessing the property.

Exercise time!

1. In folder **11-oop**, create file **oop-exercise-02.py**.
2. Create User class with **name** and **password** fields.
3. The **name** field should be accessible and the **password** field should be mangled and accessed through a **property**.
4. The initialization method should accept only the **name** field.
5. The **password's setter** should check if the password is at least 6 characters long, if it is less then it should extend the provided value to 6 characters by adding appropriate number of "#" characters ('ab' -> 'ab####'). If the password is 6 or more characters long then it should not modify it.
6. The **getter** should return the **password** in an encrypted format, that is replacing all letters except first and last with the "*" character ('password' -> 'p******d').
7. The **deleter** should delete the password.

Value and reference

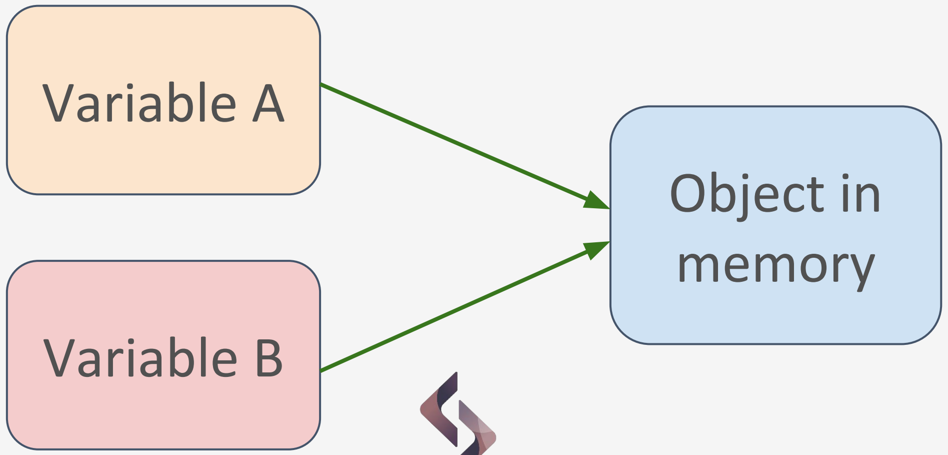
When we assign an object to a variable name, we create a **binding between them called reference**.

When we assign a new object to an existing name, the existing binding disappears and a new binding is created.

Multiple names can be bound to one object. We can call and modify the object using any of the bindings.

Value and reference

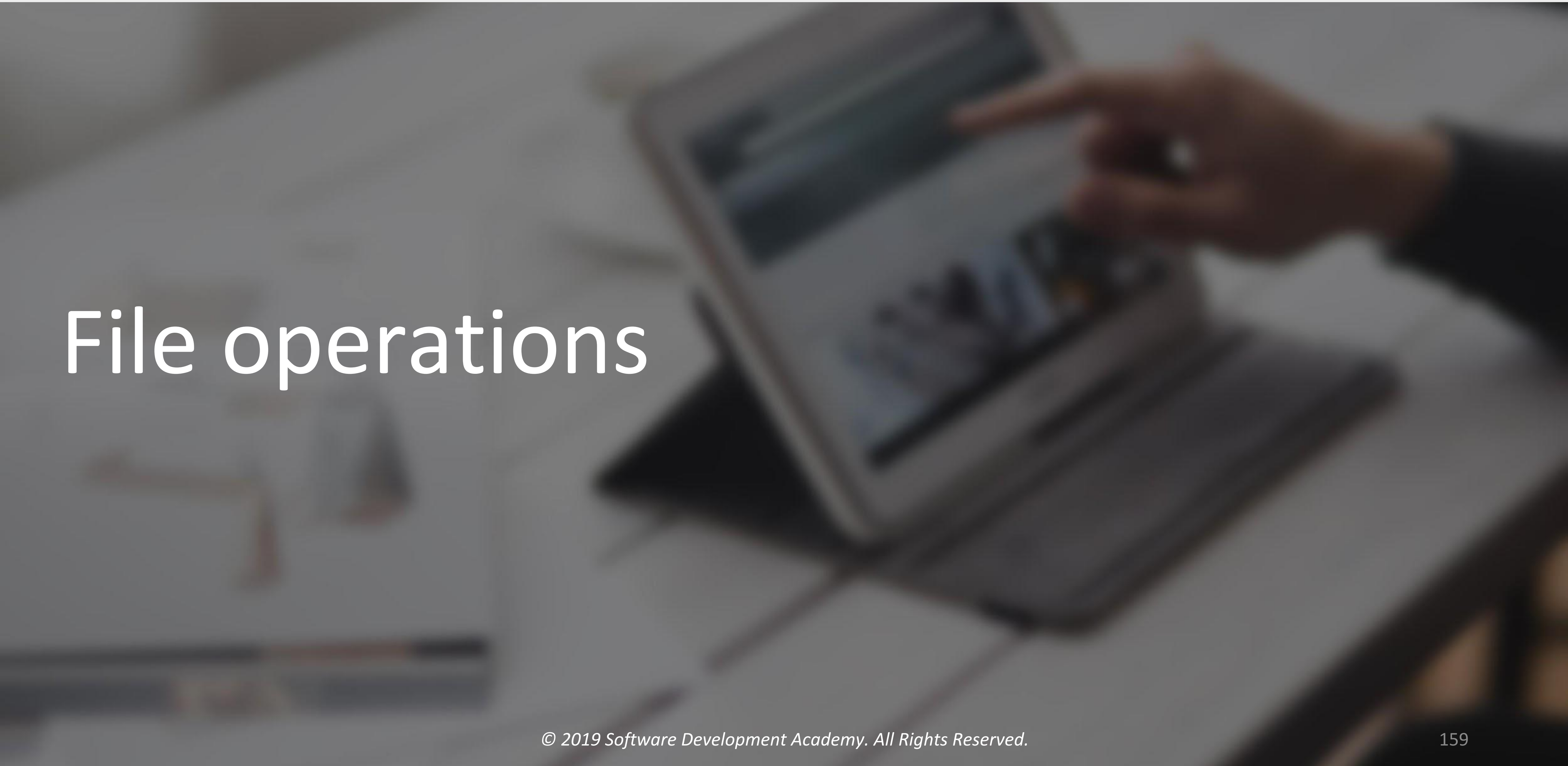
We can change state of the object using one variable and the change is reflected in all other variables, because it is the object itself that was changed and variables are just references to that object.



Value and reference

By changing object state using one reference (dog_a), the changes are present in the other reference (dog_b).

```
class Animal:
    ...

dog_a = Animal()
dog_b = dog_a
print(dog_a.name)  # prints out 'Jenna'
print(dog_b.name)  # prints out 'Jenna'

dog_a.name = "Changed value!"
print(dog_a.name)  # prints out 'Changed value!'
print(dog_b.name)  # prints out 'Changed value!'
```



File operations

File operations

To open a file in Python we must use the `open()` function, it returns a **file object**. File objects contain methods and attributes that can be used to collect information about the file and manipulate it.

The name attribute tells us the name of the file that the file object has opened.

We must understand that a **file** and **file object** are two wholly separate, yet related things.

File modes

Files have special modes that describe how a file will be used:

- "r" - read mode (default),
- "w" - write mode,
- "x" - exclusive creation, fails if the file already exists,
- "a" - open for writing, appending to the end of file (if file exists),
- "b" - binary mode,
- "t" - text mode (default),
- "+" - open file for updating (reading/writing).

Reading a file

To read a file we must use the `open()` function passing path to the file we want to open as parameter. We can then iterate every line of the file using the `enumerate()` function passing file object as parameter. It is advised to remove end of line character in every line with `.rstrip()` function before printing it out.

```
with open("data/example.txt") as f:  # f is a file object
    for i, line in enumerate(f):  # iterate every line of the file
        if i == 0:  # skip first line
            continue
        clean_line = line.rstrip()  # remove "\n" - end of line character at
the end of each line
        if clean_line == "":  # skip empty lines
            continue
        print(clean_line)
# f.close() is called automatically when code exits "with open()" block
```

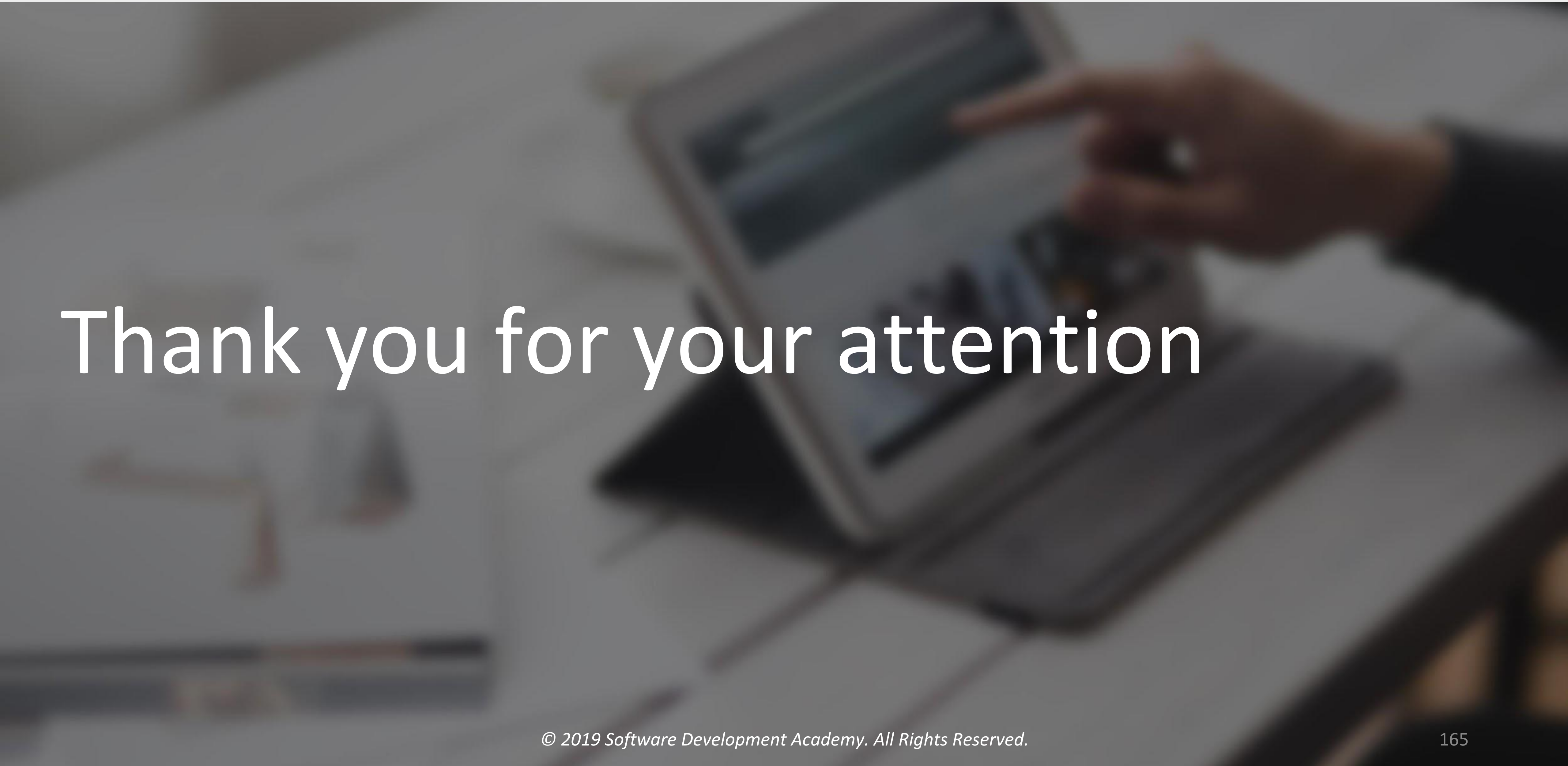
Writing to a file

To write to a file we must use the `open()` function passing path to the file we want to open as parameter. We can then write lines to the file using the `write()` function passing desired text to be written. We must explicitly specify end of line character if we want to write new lines with each loop iteration.

```
with open("data/sample.txt", "w+") as f:  # open file in write mode
    for i in range(10):
        f.write(f"This is line number {i}\n")  # write to file, adding new line
with each iteration
```

Exercise time!

1. In PyCharm, create folder **12-file-operations**.
2. Inside that folder, create file **file-operations-exercise.py**.
3. Write an application that will count how many times a word has occurred in the file and will calculate total words in the file and save the results to a new file.
4. Be careful not to count a word and a non word character such as a comma as one word (e.g. “Hello, World!” should count 2 words total, one “Hello” and one “World”).
5. Be careful not to be case sensitive (e.g. “Hello hello” should count 2 words total, two “Hello”).



Thank you for your attention