

1 Introduction to Python

Python is an object-oriented high-level language that was developed in the late 1980. as a scripting language by Guido van Rossum. The name of the language comes from a Monty Python show. It is known for its simple and intuitive syntax – it is easily picked up as a first programming language, even by non-computer-scientists. However, it is also suitable for more complex tasks – currently it is amongst the most popular programming languages. It is an open-source (free) software, available for all major operating systems. A program written on one system can be run without modifications on other operating system (**portability**).

Python has two major version (Python 2 and Python 3). The differences are subtle but the two versions are not compatible with each other. Python 3 broke backward compatibility to get rid of the historic clutter and make the language more readable. During the course, Python 3.x is used (current version available: Python 3.9).

A **program** is a sequence of instructions that define how to perform a calculation. The computation may be a mathematical problem, for example, solving an equation, but also a symbolic computation – finding a pattern in text or processing an image file. Programming can be interpreted as breaking a task into smaller subtasks – a series of programmable steps to be preformed by a computer, just like a recipe to cook a soup is a list of preparation steps.

The most important skill of a computer scientist is problem solving, which is an ability to formulate problems, think about the solution and express the solution clearly in a form of an algorithm. In this course programming is more a means to an end – solving an engineering problem.

1.1 Installation

Standard Python installation can be downloaded from: <https://www.python.org/> – it contains the Python interpreter alongside built-in and standard libraries¹.

Python interpreter with a set of libraries and a MATLAB-like, a more advanced editor (IDE, Integrated Development Environment) and a local version of Jupyter Notebooks can be installed with Anaconda: <https://www.anaconda.com/products/individual>. A very basic version of packages (instead of full Anaconda), Miniconda: <https://docs.conda.io/en/latest/miniconda.html> – no Jupyter Notebook, no IDE, only python interpreter in the basic version.

1.2 Running Python

Python programs are not compiled into machine code, they are run by the **interpreter**. The advantage of that approach is that the program can be easily and quickly debugged². On the other hand, interpreted code do not produce stand-alone applications, thus a Python script (program) can be run only on a computer which has Python interpreter installed.

There are several ways to run a Python script:

- an interactive mode – commands typed directly in a shell,
- a script mode – commands stored in a file and run by the Python interpreter from an interactive mode or an IDE,
- Jupyter Notebook.

Most of the editors and methods of running the Python code enable code autocompletion with a Tab key.

¹See more help about installation: <https://www.python.org/about/gettingstarted/#installing>. During the class, Jupyter Notebook or some IDE might be useful, remember to install those as well. Anaconda has it all, so it may be simpler to choose that method

²debugging is a process of finding programming errors in code; programming errors are called **bugs**

1.2.1 Interactive mode

One of the methods of running Python is using it as a console application, either via `python` or `ipython` app (Figure 1.1). Python interpreter can be started from command line³ by typing `python`. After starting the interpreter, Python version is shown, and then, the prompt command (`>>>`) appears, which indicates that the program is ready for the programmer to enter the code. A line of code is confirmed with a **Return (Enter)** key and the result is displayed immediately. This way of running Python is called an **interactive mode**. The interactive mode works well for a quick check of a function or evaluating an expression, however, it is not very useful for longer computations. It is difficult to correct an error in a multiline call.

In interactive mode, the last printed expression is assigned to a variable `_` (example 1.1). The variable should be treated as read-only and no value should be assigned to it – in that case a local variable would be created with the same name and it would mask the built-in variable.

Example 1.1. Using the result of the previous expression

```
1 a = 2+2
2 a * 8
3 new_value = _ + 10
```

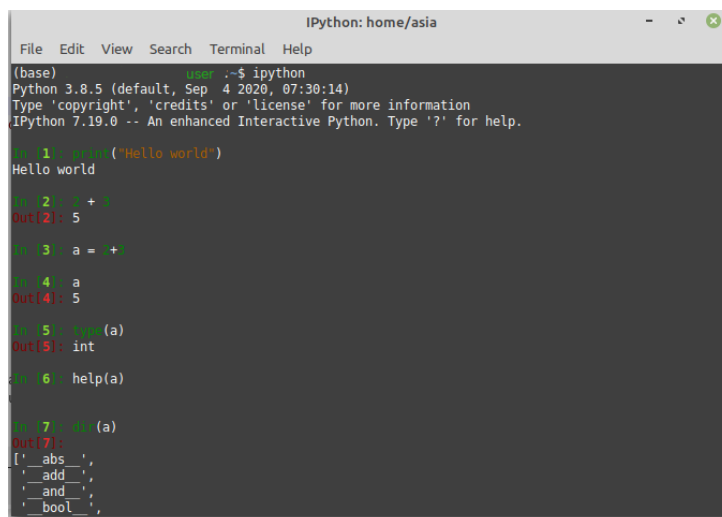


Figure 1.1. Trying out mathematical operations in IPython

1.2.2 Script mode

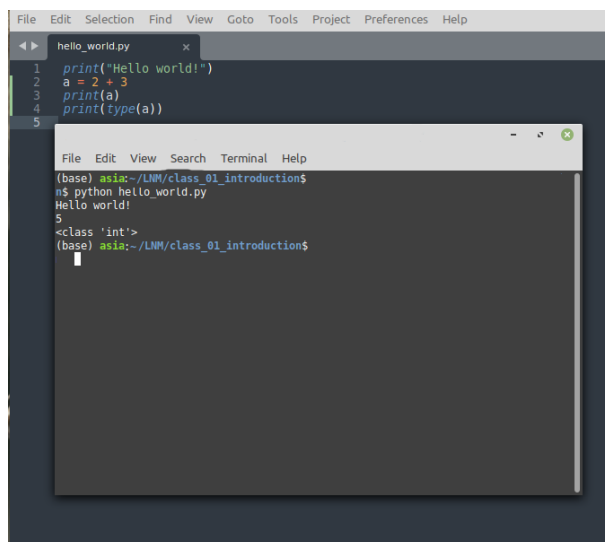
Interactive programming is useful for learning and quickly developing simple code, complex projects, however, require more complex programs. Those programs, which include many lines of Python code, are saved to text files with `.py` extension⁴. The main difference between running the same code in the interactive mode and in the script mode is that the interpreter displays the results instantly, but running a script has no visible effect. To see the results, a `print` statement is required (section 1.13).

³cmd in Windows

⁴Any extension will work as long as the file contains Python code, but to indicate that the file contains Python code, `.py` is used in general. However, you can try saving the commands to a `.txt` file and run it with the Python interpreter. The file can be edited with any text editor (Notepad is the simplest, Notepad++ or Sublime are more complex and include syntax highlighting), but programmers usually use IDEs (Integrated Development Environment)

Python code can be typed in any text editor, however a convenient function is code highlighting – text editors recognise Python code and highlight sections of the program and keywords with different colours (**syntax highlighting**), making it easy to understand the structure of the code.

To run your program (called **a script**), use the magic `%run` command in IPython environment: `%run script.py` or launch it directly from the standard interpreter: `python script.py` (Figure 1.2). The advantage of using IPython environment is that the variables used in the program (script) can be examined and used after the program run has completed. Calling Python directly will run program and then terminate, so, if any output or important results are necessary, they should be saved to a file or printed to the console.



The screenshot shows a code editor window titled 'hello_world.py' with the following code:

```
1 print("Hello world!")
2 a = 2 + 3
3 print(a)
4 print(type(a))
5
```

Below the code editor is a terminal window. The terminal shows the command `python hello_world.py` being executed, resulting in the output:

```
(base) asia:~/LNM/class_01_introduction$
n$ python hello_world.py
Hello world!
5
<class 'int'>
(base) asia:~/LNM/class_01_introduction$
```

Figure 1.2. Running a script from a command line

1.2.3 Jupyter Notebook

Somewhere between the interactive and script mode **Jupyter Notebook** appears. **Jupyter Notebook** is an interactive computing environment that allows to run small fragments of code as well as more complicated programs, and gives immediate results. It is very convenient for trying out bits of code or processing data in a structured way. Apart from the code, description provided as an ordinary text, mathematical equations and images can be included in the Notebook (Figure 1.3). Jupyter Notebook uses the notation called **Markdown** to create a simple syntax for structuring the text⁵.

Notebook contains the inputs and outputs of the interactive session as well as text describing the code (not meant for execution). The notebook documents are saved with an extension `.ipynb`. All actions in the notebook can be performed using the mouse, however keyboard shortcuts sometimes speed up the work, most important shortcuts being:

- **Shift + Enter** – run cell – executes the cell, shows output and jumps to the next cell;
- **Esc** – go to command mode – navigate around the notebook using keyboard shortcuts, allows adding new cells with A and B keys; recognized by blue line on the left side of the cell;
- **Enter** – edit mode – edit text (code) in cells; recognized by the green line on the left side of the cell;

⁵More information on Markdown: <https://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/Working%20With%20Markdown%20Cells.html>

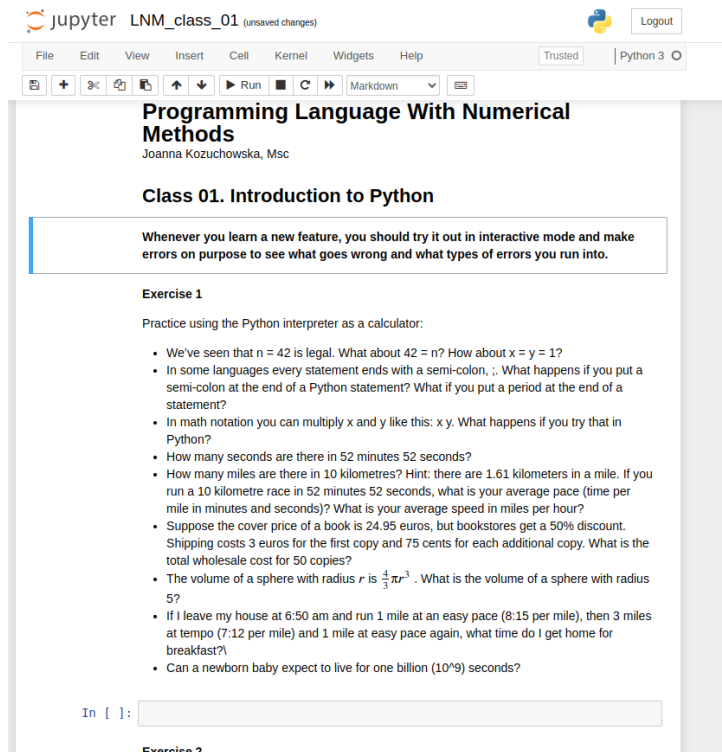


Figure 1.3. Jupyter Notebook example

- **Y** – (command mode) changes cell to code;
- **M** – (command mode) changes cell to markdown;
- **X** – (command mode) cuts selected cell;
- **Z** – undos the last cell operation, either in Edit or Command mode.

More information about the application can be found at <https://jupyter-notebook.readthedocs.io/en/stable/notebook.html>. Jupyter Notebook is also a part of Anaconda environment.

Apart from functions available in Python, there are few commands available also in IPython, called magic commands:

- **%magic** – print information about the magic functions,
- **%%function** – shows help for the function
- **%time** – time execution of a Python statement or expression,
- **%who** – prints all interactive variables,
- **%whos** – like **who**, but gives extra information.

Help on the function that cursor is currently at can be also accessed with pressing **Shift + Tab** keys.

1.3 First program

Usually, the first program to write in a new language is called **Hello, world** – a program which displays a single line, saying "Hello, world". In Python, it requires a **print** function to display the sentence on the screen, and a definition of the character string, which, in Python is defined with double

or single quotes. `Hello, world` in Python will look as in example 1.2. If you try this example in the Python interpreter or run a script including this line, the text inside the `print` function paranthesis will appear – without the quotes.

Example 1.2. Hello, world in Python

```
1 print("Hello, world!")
```

1.4 Values, types and variables

A **value** is one of the basic things that a program works with. A value can be a letter, a number or a sequence of numbers (or letters). Those values belong to different **types**: `1` is an **integer**, but `1.0` is a **floating-point number** and `"1.0"` or `"Hello"` are **strings** – sequences of characters strung together. A type is a category of a value. To check the type of a value, use `type` function, like in example 1.3. In return, you get the name of the **class**; a class is a definition of the object. In Python, both class and type mean more or less the same.

Example 1.3. Checking the type of a value

```
1 a = 2.0
2 type(a)
```

1.5 Simplest built-in data types

The simplest built-in data types in Python are:

- **integers** – whole numbers (`1`, `2`, `6700000`),
- **floats** – numbers with decimal points (`3.14`) or exponents (`2.0e6`),
- **booleans** – `True` or `False` values,
- **strings** – sequences of text characters.

Each type is handled differently by the computer and has a set of rules regarding its use. Those types can be also used to build more complicated structures (introduced later).

Python is an **object-oriented** language, and everything (every piece of data and value) is an object. An object can be interpreted as a box that contains a piece of data. The object has a type (such as integer or string) which determines what operations can be done on the data. The data type determines also if the content of the object can be changed (**mutable** and **immutable** types).

1.6 Variables

One of the most important features of a programming language is the ability to process **variables**. A variable is a name that represents a value of a certain type stored in computer memory. In most programming languages, types are predefined and a variable can store only one type of information throughout the run of the program. This is not the case in Python, which is a **dynamically typed** programming language. In a dynamically typed language there is no need to declare a type of a variable. Moreover, the type of the variable can change during the run of the program (which can be a plus, but also a drawback – if you type the name of the variable incorrectly there is no warning about the type changing and the old value is overwritten).

Example 1.4. An assignment statement

```
1 a = 1
2 a = a * 1.0
3 type(a)
```

An **assignment statement** `a = 1` in example 1.4 creates an association between the name `a` and an integer value 1 stored in the memory: creates a new variable (a label for a value stored in memory) and gives it a value. The `=` is referred to as an **assignment operator**. The next statement, first, evaluates the expression `a*1.0` and then, associates the value with the name `a`. The original association with integer 1 is now destroyed and `a` refers to a floating point value. A variable is created when a value is assigned to it. The same name of the value can be present on both, the left and right side of the equation, because first, the right-side expression is evaluated based on the current values of variables and then the updated value is assigned to a name.

There are few rules for naming variables. Variable names:

- can only contain alphanumerical characters and an underscore,
- cannot start with a digit,
- are case-sensitive, so variable `A` is not the same as `a`,
- should start with lower case and use an underscore to separate words instead of spaces,
- variables that start with an underscore have a special meaning (convention will be introduced later),
- should be chosen to be meaningful (but not too long) – they document what the variable is used for,
- should not be the same as Python keywords (keywords are usually highlighted, if syntax highlighting in the editor is available),
- if invalid, cause a **syntax error**,
- should not contain lowercase letter `l` and uppercase `O` or uppercase `I` (i) – they can be easily confused with the numbers 1 and 0.

Variables must be created before they are used. If a variable does not exist or was misspelled, Python reports an error.

In Python, **variables are just names**. An assignment operation does not copy a value, it attaches a name to the object that contains the data. The name is a **reference** to a place in memory where the data is stored rather than the value itself.

1.6.1 Type conversions

If an arithmetic operation involves different variable types, before the operation is carried out, usually, the numbers are automatically converted to a common type. For example, while adding `2 + 3.0`, first, 2 is **casted** (the type of the value is converted) to `2.0`. Only then, the addition is done. Sometimes the casting is not possible, as in `"a" + 7`.

Type conversion can be also done explicitly, by casting a variable to a different type with a function:

- `int(a)` to convert `a` into an integer; conversion from float to integer is carried out by truncation, not a round off. `int` can't handle a conversion from `str` containing decimal points or exponents.
- `float(a)` to convert `a` into a floating point number. If you try to convert something that cannot be interpreted as a number, you'll get an error.

- `complex(a, b)` to convert `a` and `b` into a complex number with real part `a` and imaginary part `b`⁶,
- `str(a)` to convert `a` into a string.

`True` and `False`, which are boolean values are often casted to `int`. `True` is represented with 1 (or any other non-zero value) and 0 can be interpreted as `False`.

1.6.2 Updating variables

A commonly used type of reassignment of a variable is an **update** – the new value of the variable depends on the old, e.g. `x = x + 1`, which means “get the current value of `x`, add 1 and update `x` with the new value”. The variable that does not exist, cannot be updated (an attempt to do so, will raise an error). Before a variable can be updated, it needs to be **initialized**, usually with an assignment statement. An empty variable (empty list, set or tuple) can be initialized as well. Updating a numerical variable by adding 1 is called **increment** and subtracting 1 is called **decrement**.

1.7 Basic arithmetic

Python provides **operators** which are special symbols that represent computations like addition or subtraction. The usual arithmetic operators are supported in Python:

+ addition,

- subtraction,

* multiplication,

/ floating point (decimal) division,

** exponentiation,

% modulus (remainder) division – divides two numbers and returns a remainder,

// integer (truncating, integer) division – divides two numbers and rounds down to an integer;

Be careful with the exponentiation operator. In some other languages (or in Microsoft Excel), exponentiation is defined with a `^` character (e.g. `6^2 = 36`), but in Python `^` is a bitwise operator XOR, and `6^2` would give a result 4. Some of the operators are also defined for strings and sequences (lists and tuples).

Python also has **augmented operators** or **compound assignment operators** (which may be familiar to C programmers). The augmented operators and equivalent arithmetic expressions are shown in example 1.5

Example 1.5. Augmented operators

```
1 a += b      # a = a + b
2 a -= b      # a = a - b
3 a *= b      # a = a * b
4 a /= b      # a = a / b
5 a **= b     # a = a ** b
6 a \%= b     # a = a \% b
```

⁶`int`, `float`, `complex` will also work for strings if the string variable represents a valid number

1.7.1 Order of operations

When an expression contains more than one operator, the order in which the operations are evaluated depends on the **order of operations**. Order of mathematical operators in Python follows mathematical convention. Use an acronym **PEMDAS** to remember the rules:

- **P**arentheses have the highest priority and can be used to force evaluating an expression in the order that you want,
- **E**xponentiation,
- **M**ultiplication and **D**ivision,
- **A**ddition and **S**ubtraction,
- operators with the same precedence are evaluated from left to right.

1.7.2 Mathematical functions available in standard Python library

Standard Python library includes only a few basic mathematical functions:

- `abs(a)` – absolute value of `a`,
- `max(sequence)` – largest element in a `sequence`,
- `min(sequence)` – smallest element in a `sequence`,
- `round(a, n)` – round `a` to `n` decimal places,
- `pow(a, n)` – a^n , `a` raised to the power of `n`,
- `cmp(a, b)` – compare `a` and `b`, returns -1 (`a<b`), 0 (`a==b`) or 1 (`a>b`).

To use more advanced functions, additional modules, e.g., `math` or `numpy` are needed.

A **module** is a file that contains a collection of related functions. To use a function from a module, an import statement is necessary (example 1.6). To access one of the elements of the module – a function or a variable, the name of the module and the name of the element needs to be specified, separated by a dot (**dot notation**, example 1.7).

Example 1.6. Importing a math module

```
1 import math
2 # or, to use an alias 'm' instead of a full name
3 # import math as m
4 # sinus of a pi angle
5 sinpi = math.sin(math.pi)
```

Example 1.7. Dot notation

```
1 sinpi = math.sin(math.pi)
```

1.8 Expressions and statements

An **expression** is a combination of values, variables and operators. A value on its own is considered an expression (and so is variable). When the expression is typed in the interpreter, it is **evaluated** – the value of expression is found. A **statement** is a unit of code that has an effect, e.g. displaying a value or creating a variable (example 1.8). When a statement is typed, it is **executed** – the interpreter does whatever the statement says. Generally, statements do not have values.



Example 1.8. Expressions and statements

```
1 30          # expressions
2 a
3 a + 15
4 a = 5 # statements
5 print(a)
```

1.9 Comments

As your programs grow bigger, it is a good idea to describe particularly tricky lines of code with notes explaining the flow of the script or solution used. Comments in Python are introduced with a `#` character (called *hash*, *sharp* or *pound*) and are ignored by the Python interpreter. Comments may appear at the end of a line or be a line of itself (example 1.9). Everything from `#` to the end of the line is ignored and has no effect on program execution. Good variable names reduce the need for comments (code becomes self-explanatory). Longer comments which are part of code documentation and often span multiple lines, can be introduced (and then closed) with triple quotes (single or double).

Example 1.9. Commenting the code

```
1 """
2 Long introduction to what the code does
3 Can span multiple lines
4
5 Even more lines
6 """
7
8 print("hello world") # comment in the middle of the line
9 #comment at the beginning of the line
```

1.10 Whitespace sensitivity

Python is whitespace sensitive and the indentation (spaces or tabs) affects how the files are interpreted. Introducing a space at the beginning of the line will cause an error. Every block of code which modifies the flow of the program (conditional statements, loops, defining functions or classes, basically every statement that ends with a colon) requires an indented block of code. Every statement in the block will have the same offset (indentation) from the beginning of the line.

1.11 Importing modules

Using libraries that are not built-in Python libraries (standard libraries – provided with the Python installation and additional libraries downloaded by the user) require an **import statement**. There are several ways to load the packages:

- `import package_name` – imports every element in `package_name`, which then can be accessed via dot notation: `package_name.function_name()`;
- `import package_name as pn` – same as above, but an alias is used in the dot notation instead of the whole name: `pn.function_name()`;
- `from package_name import *` – imports every element in `package_name`, which then can be accessed using their name: `function_name()` – may overwrite some functions and variables existing in the namespace;

- `from package_name import function_name1, function_name2` – imports only some functions from the package, which can be accessed with their name: `function_name1`, `function_name2`.

1.12 Keyboard input

If an input from a user is needed (instead of hard-coded values), a built-in `input` function will be useful. The function stops the program and waits for the user to type something. Then the user presses **Enter**, the program resumes and interprets the typed value. If there is no input from the user (not even an **Enter**), the program will not execute any further. A return value from an `input` function is a **string** variable, doesn't matter if it is only numbers or only letters, **it is always a string**. If the input value is an operand of a mathematical operation (number to be processed), it needs to be cast to a numerical type (example 1.10). If a user types something which cannot be interpreted as a numerical value, the error occurs and the program stops.

Example 1.10. Input from the user

```
1 x = input("Type x: ")
2 print("Here's x+5: ", float(x) + 5)
```

It is recommended to state what type of input we expect from the user. The message can be included as an optional argument in the `input` function (see Listing 1.10, line 1). Another way is to print the message about the expected input separately (listing 1.11).

Example 1.11. Input from the user with `print` function

```
1 print("Type x: ")
2 x = input()
3 print("Here's x+5: ", float(x) + 5)
```

1.13 Output

To put some text on the console or in the file, the `print` function is needed (example 1.12).

Example 1.12. `print` function

```
1 print(value1, value2, ..., sep=" ", end="\n", file="sys.stdout", flush=False)
```

Values provided to the function are values that need to be printed. To add a string (text) to the printout, put it in single or double quotes. By default, the values are printed to the standard output (console) and are separated with a space (can be changed by modifying `sep` argument value). Empty argument list, `print()`, prints an empty line.

1.14 Debugging

As mentioned earlier, **debugging** is a process of tracking down the errors (bugs) in a program. Three kinds of errors occur in a program:

- syntax errors – refer to the structure of the program and its rules, e.g. no matching parentheses or lack of `:` character. After encountering the syntax error, Python displays the error message and quits (the program cannot be run). The syntax error is reported when Python cannot understand the source of the program (it won't even try to run it).
- runtime errors – do not appear until after the program has started running. Runtime errors are also called **exceptions**, because they usually indicate that something exceptional has happened. Runtime error is reported when something goes wrong while the program is executing

- semantic errors – related to meaning. If a program contains a semantic error, it will run without generating an error message, but it will not do the expected thing – it will do the thing it was programmed to do, which may be not what a programmer had in mind. Fixing those errors is not easy as it requires working backward by looking at the output and trying to figure out what went wrong.

1.15 Useful links

- Recommended read:
 - <https://docs.python.org/3/tutorial/introduction.html>
 - <https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-long-complex>
 - [additional] Is Python interpreted or compiled language? https://nedbatchelder.com/blog/201803/is_python_interpreted_or_compiled_yes.html
 - check out a file `class_01_links.txt`
- Allen B. Downey, "Think Python":
 - Introduction: <http://greenteapress.com/thinkpython2/html/thinkpython2002.html>
 - Variables: <http://greenteapress.com/thinkpython2/html/thinkpython2003.html>
- online Python interpreter: <https://repl.it/languages/python3>
- Jupyter Notebooks (online): <https://cocalc.com/doc/jupyter-notebook.html>, Google Colab;

References

- Downey, A. B. (2016). *Think Python. How to Think Like a Computer Scientist*. O'Reilly Media, 2nd edition.
- Lutz, M. (2013). *Learning Python*. O'Reilly Media, 5th edition.
- Python (2020). Python 3.8.5 documentation. <https://docs.python.org/3/> [Accessed 10 December 2019].