02. Python basics

September 17, 2025

1 2. Python basics

1.1 2.1 Comments

Comments are parts of a program which are useful for anyone reading the code, but ignored completely by the interpreter. Python considers a comment anything that comes after a `#` symbol.

Comments can: * explain assumptions * explain important decisions * explain important details * explain problems you're trying to solve * explain problems you're trying to overcome in your program, etc. * temporarily disable a part of the program

This is useful for readers of your program so that they can easily understand what the program is doing. Remember, that person can be yourself after six months!

script1.py

```
# These exact names are also used in script2.py
# If you change them make sure you update both places
statuses = [
    'new',
    'open',
    'closed',
]
```

1.2 Exercises 2.1

1. Add a comment in `first_script.py` explaining what it does. Run the program again.
2. Try commenting out the first line (the one where variable `name` is initialized) and run the program. What happens?

1.3 2.2 Errors

Every statement we write in a computer program is an instruction / command for the computer to execute. When the computer cannot do that, it will *throw an error*. These errors can happen for various reasons, such as syntactical mistakes, logical errors, or unexpected situations during runtime. When an error is thrown, the program stops completely.

Errors can be dealt with through exception handling, but we will come back to this subject later in this course. For now, we will learn to interpret error messages and change the code to prevent error occurrence.

In the exercise above, you noticed the following error:

[2]:
```
print(some_variable)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[2], line 1
----> 1 print(some_variable)

NameError: name 'some_variable' is not defined
```

The interpreter gives us several important details when an error occurs, in order to help us debug it: 1. The error type: `NameError`

This is the name of the exception class. There are several built-in exceptions in Python, besides `NameError`. We will discuss them when we encounter them. 2. The error message: `name 'some_variable' is not defined`

This message should help us understand what went wrong. In this case, the variable named `some_variable` has not been previously defined, which is why Python is unable to print its value. 3. The traceback

Shows the line that triggered the error and all the function calls that led to the erroneous statement. It is especially useful in more complex programs.

A `NameError` will also occur if we forget to enclose a string in quotation marks:

[3]:
```
print("hello")  # this one prints the text `hello`
```

```
hello
```

[4]:
```
print(hello)  # this one assumes a variable named `hello` has been defined
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 print(hello)  # this one assumes a variable named `hello` has been␣
 ↪defined

NameError: name 'hello' is not defined
```

Another error you might see quite often while learning Python is `SyntaxError`. This happens when something you wrote isn't syntactically correct. Modern IDEs will try their best to prevent you from making syntax errors, but they can still occur. Let's see some examples:

[5]:
```
# different quotation marks used around the string
print("hello')
```

```
  Cell In[5], line 2
    print("hello')
          ^
```

```
SyntaxError: unterminated string literal (detected at line 2)
```

[6]:
```
# missing parenthesis
print("hello"
```

```
  Cell In[6], line 2
    print("hello"
                 ^
_IncompleteInputError: incomplete input
```

1.4 Exercises 2.2

1. The following lines of code contain some errors. Copy them into a Python file, execute it, and then solve the issues one by one.

```
print(hello world)
message = hello
print(message)
print("this is a string)
print('this is another string")
```

1.5 2.3 Variables

1.5.1 What are variables?

Think of variables as labels that you stick on boxes with data. These boxes have names (or labels) that you assign to them, making it easy for you to refer to the data they contain.

In Python, you can create a variable simply by choosing a name and assigning a value to it using the equal sign (`=`). For example:

[7]:
```
message = "hello world"
```

In this example, `message` is the name of the variable, and `"hello world"` is the value assigned to it. Now, whenever we refer to `message` in our code, Python knows that we're talking about the text `"hello world"`.

1.5.2 Variable types

As you noticed, there is no need to specify the type of the variable when declaring it. **Variables are not bound to fixed types**. This is because Python is a dynamically typed language, meaning that variable type is determined at runtime. However, Python is also strongly typed, which means variables do have a type and that the type matters when performing operations on a variable (more on this later). You can always inspect the type of a variable using the built-in function `type`:

[8]:
```
type(message)
```

[8]:
```
str
```

Variable type can also change at runtime, although this is not a common operation:

[9]:
```
message = 0
```

[10]:
```
type(message)
```

[10]:
```
int
```

We will dive into the specifics of each data type in the following chapters.

1.5.3 Variable names

When choosing names for your variables, there are a few rules to follow:

1. Variable names can contain letters (a-z, A-Z), digits (0-9), and underscores (_), but they cannot start with a digit.
2. Variable names cannot contain spaces or special characters like !, @, #, $, %, etc.
3. Variable names are case-sensitive, meaning `message`, `Message`, and `MESSAGE` would be considered different variables.
4. Choose descriptive names that reflect the purpose of the variable.

1.5.4 Using variables

Once you've assigned values to variables, you can use them in your code wherever you need that value. For example:

[11]:
```
x = 10
y = 5
z = x + y
print(z)
```

```
15
```

In this example, we added the values of `x` and `y` and stored the result in `z`. Then, we printed the value of `z`, which is `15`.

1.6 Exercises 2.3

1. In this exercise, you'll create a simple daily activity tracker. Copy the lines below into a Python file, and update `current_activity` to represent different activities done throughout the day.

```
# Define the initial activity
current_activity = "Morning jog"

# Printing out morning activity
print("Morning:")
print(current_activity)

# Update activity for the afternoon
```

```
# Printing out afternoon activity
print("Afternoon:")
print(current_activity)

# Update activity for the evening

# Printing out evening activity
print("Evening:")
print(current_activity)
```

1.7 2.4 Numbers

Python has multiple ways of storing numbers, which can be used alternatively depending on the intended purpose of the number we're using. The most commonly used numeric data types in Python are:

- **Integers (type `int`)**

  Integers are whole numbers without any decimal point. They can be positive, negative, or zero. For example: `5`, `-10`, `0`. If you were storing the population of a country, the number of apartments in a building or counting the number a times you encountered a certain value in a sequence, you would be using an integer number.

- **Floating-Point Numbers (type `float`)**

  Floating-point numbers, or floats, represent real numbers with a decimal point. They can also be expressed using scientific notation. For example: `3.14`, `2.5e2` (which is equivalent to 250.0). If you were storing the area of a country, measuring the distance between two cities or calculating the perimeter of a terrain, you would likely use a floating-point number.

Numbers can be assigned to variables, or can be used literally in a program:

[12]:
```
my_int = 10
my_float = 4.5
print(my_int + 2)
```

```
12
```

Above, we used both variables (`my_int` and `my_float`) and numeric literals (10, 4.5 and 2) to refer to numbers. We call the numbers 10, 4.5 and 2 here *numeric literals*, because they are the actual numbers, not variables storing these values.

1.7.1 Type Conversion

You can convert between different numeric types using type conversion functions. For example:

[13]:
```
new_float = float(my_int)
print(new_float)
```

```
10.0
```

[14]:
```
new_int = int(my_float)
print(new_int)
```

```
4
```

1.8 Exercises 2.4.1

1. Declare a variable called `my_age` and assign it your current age as an integer. Print the value.
2. Declare a variable called `distance` and assign it a length in meters as a float. Print the value.
3. You decide you actually want to store the distance as an integer. Convert `distance` created above to `int` and assign the result to the same variable. Print the new value.

1.8.1 Arithmetic operations

Most statements that you write will contain expressions. A simple example of an expression is `10 + 2`. An expression can be broken down into *operators* and *operands*.

Operators are special symbols, combinations of symbols, or keywords that designate some type of *computation*. Operators require some data to operate on and such data is called *operands*. In this case, 10 and 2 are the operands.

Python provides several operators for performing arithmetic operations on numbers. These operators include:

- **Addition (`+`)**: Adds two numbers together.
- **Subtraction (`-`)**: Subtracts one number from another.
- **Multiplication (`*`)**: Multiplies two numbers.
- **Division (`/`)**: Divides one number by another. Always returns a float.
- **Floor Division (`//`)**: Divides one number by another and returns the integer part of the result.
- **Modulus (`%`)**: Returns the remainder of the division of one number by another.
- **Exponentiation (`**`)**: Raises one number to the power of another.

[15]:
```
result = 10 - 2
print(result)
```

```
8
```

An expression can also be sent to the `print` function, without previously assigning it to a variable:

[16]:
```
print(10 * 2)
```

```
20
```

An interesting feature in Python is that it defines two division operators: `/` will always return a float, while `//` will return the quotient.

[17]:
```
10 // 4
```

[17]: 2

[18]:
```
10 / 4
```

[18]:
```
2.5
```

Python fully supports mixed arithmetic, which means a float and an integer can be used as operands of a binary arithmetic operator. The result will be a floating point number:

[19]:
```
4.5 * 2
```

[19]:
```
9.0
```

1.8.2 Evaluation Order

If you had an expression such as `2 + 3 * 4`, is the addition done first or the multiplication? Mathematics rules tell us that the multiplication should be done first. This means that the multiplication operator has higher precedence than the addition operator. The same rule has been implemented in Python. For arithmetic operators, they will be evaluated in the following order:

1. Exponentiation (`**`)
2. Multiplication (`*`) and divisions (`/`, `//`, `%`)
3. Addition (`+`) and substraction (`-`)

Operators on the same priority level (e.g. multiplication and floor division) will be evaluated in order, from left to right.

[20]:
```
10 + 2 * 5 ** 2
```

[20]:
```
60
```

Changing the order of evaluation If we need to change the order in which operators are evaluated, we can add parentheses to group parts of the expression:

[21]:
```
((10 + 2) * 5) ** 2
```

[21]:
```
3600
```

Parentheses can also be added to make an expression more readable, without changing the order of evaluation:

[22]:
```
10 + 2 * (5 ** 2)
```

[22]:
```
60
```

1.9 Exercises 2.4.2

1. Declare a variable called `my_age` and assign it your current age as an integer. Declare a variable called `friend_age` and assign it the age of a friend as an integer. Calculate the age difference between you and your friend and store it in a variable called `age_difference`. Print the age difference.
2. Romania's GDP in 2022 was 284 billion euros. In the same year, Romania's population was 19.5 million. Define two variables `gdp` and `population` and store these values. Compute and display Romania's GPD per capita in 2022.

3. Declare a variable called `celsius_temp` and assign it a temperature in Celsius as a float. Convert the temperature to Fahrenheit using the formula: `Fahrenheit = (Celsius * 9/5) + 32`. Store the result in a variable called `fahrenheit_temp`. Print the Fahrenheit temperature.
4. Declare a variable called `distance_meters` and assign it a length in meters as a float. Convert the length to feet using the conversion factor: 1 meter = 3.28084 feet. Store the result in a variable called `distance_feet`. Print the distance in feet.
5. Define a variable called `total_minutes` and assign it a random value chosen by you between 0 and 1440. Consider this to be the number of minutes passed after midnight. What time would a digital 24h clock show? Compute and display the two values: `hour` and `minute`.

1.10 2.5 Strings

In Python, a string is a sequence of characters enclosed within either single quotes (`' '`) or double quotes (`" "`). Strings can contain letters, digits, special characters, and even whitespace. Here's an example of a string:

[23]:
```
message = "Hello, World!"
```

In this example, `"Hello, World!"` is a string. We also call these *literal strings*, because the content of the string is defined literally, between the quotes. Literal strings should always be enclosed by quotes; skipping the quotes will trigger an error, like we've already seen in the Errors chapter above:

[24]:
```
message = Hello, World!
```

```
  Cell In[24], line 1
    message = Hello, World!
                         ^
SyntaxError: invalid syntax
```

1.10.1 Creating Strings

You can create strings using single quotes, double quotes, or even triple quotes for multi-line strings.

Single and double quotes are interchangeable - there is no difference between strings created with single vs double quotes. Developers usually choose one style of quotes and use it consistently.

One case where using one type of quotes over the other is preferable are strings containing a quote. For example, the string `What's up?` cannot be enclosed in single quotes, because Python would consider the string to be terminated by the apostrophe after `What`:

[25]:
```
question = 'What's up?'
```

```
  Cell In[25], line 1
    question = 'What's up?'
                         ^
SyntaxError: unterminated string literal (detected at line 1)
```

In this case, using double quotes will solve the issue:

[26]:
```
question = "What's up?"
```

Same rule goes for double quotes enclosed inside single quotes:

[27]:
```
text = '"To be or not to be", that is the question'
```

Triple quotes are used for multi-line strings, i.e. strings that contain multiple lines. Newlines added inside the triple quotes are kept inside the resulting string:

[28]:
```
multiline_string = """This is a
multi-line string."""
print(multiline_string)
```

```
This is a
multi-line string.
```

1.10.2 Escape Sequences

Another approach to define strings that contain special characters (like the quotes that enclose strings) is using escape sequences. Escape sequences in Python are special characters that are preceded by a backslash (\) and used to represent characters that are difficult or impossible to type directly into the code. They allow you to include characters such as newline, tab, backslash, and others within strings.

Here are some common escape sequences:

- `\n`: Represents a newline character. It is used to add a new line within a string.

[29]:
```
print("Good\nMorning!")
```

```
Good
Morning!
```

- `\t`: Represents a tab character. It is used to insert horizontal tab spaces within a string.

[30]:
```
print("Name:\tAnna")
```

```
Name:    Anna
```

- `\'`: Represents a single quote character. It is used to include a literal single quote within a string enclosed by single quotes.

[31]:
```
print('What\'s up?')
```

```
What's up?
```

- `\"`: Represents a double quote character. It is used to include a literal double quote within a string enclosed by single quotes.

[32]:
```
print("He said \"Hello!\"")
```

```
He said "Hello!"
```

- \\: Represents a backslash character. Because the backslash is an escape character, in order to include it literally in a string, it has to be escaped (by itself).

[33]:
```
print("This is a backslash: \\")
```

```
This is a backslash: \
```

1.10.3 Exercises 2.5.1

1. Create two variables, `single_quote_string` and `double_quote_string`, each containing the same string, but enclosed with different quotation mark symbols. Compare the two variables (using `==`, an operator which verifies if two values are equal). Are the two strings equal?

   ```
   print(single_quote_string == double_quote_string)
   ```

2. Create a multiline string (using triple quotes) containing the lyrics of a song you like (choose a few lines). Print the string.

3. Do the same as the previous exercise, but this time use newline characters inside a single line string. Which option was easier to write?

4. Create a string that outputs, when printed, the following text. Use newlines (between lines) and tabs (after the `:` symbol).

   ```
   name:    Anna
   age: 20
   hobby:   writing
   ```

1.10.4 String manipulation

Python supports a series of operations for string objects, which we dive into next.

Checking if a substring is part of a string In Python, the `in` and `not in` operators are used to check whether a substring exists within a string or not. These operators return a boolean value (True or False) based on the presence or absence of the specified substring.

[34]:
```
text = "Python is a programming language"
word = "lang"
print(word in text)
```

```
True
```

[35]:
```
print("a" not in text)
```

```
False
```

Concatenation and multiplication You can combine two or more strings together using the `+` operator, resulting in a new string:

[36]:
```
greeting = "Hello, " + "Anna" + "!"
print(greeting)
```

```
Hello, Anna!
```

Somewhat surprising, the `*` operator also works on strings. The second operand **must** be an integer, and the result will be a new string obtained by concatenating the initial string to itself:

[37]:
```
greeting = "Hello! " * 3
print(greeting)
```

```
Hello! Hello! Hello!
```

Indexing Strings are sequences of characters. Each character in a string has a certain position, or *index*. For example, the string `"Python"` has a `P` on the first position, a `y` on the second position, and so on. Indexing always starts with position 0, so `P` would actually be on index 0, `y` on position 1, and so on. Here is a table with each character and its index in the string `"Python"`:

Index	Character
0	P
1	y
2	t
3	h
4	o
5	n

The indexing operation means retrieving a character in a string from a certain index:

[38]:
```
word = "Python"
word[0]
```

[38]:
```
'P'
```

[39]:
```
word[1]
```

[39]:
```
'y'
```

Negative indexes can also be used, and they will retrieve characters starting from the end of a string:

[40]:
```
word[-1]
```

[40]:
```
'n'
```

[41]:
```
word[-2]
```

[41]:
```
'o'
```

Slicing The slicing operation is pretty similar to the indexing one, but this time we will retrieve *slices* (multiple characters) of the string. The slicing notation is `string[start:end:step]`, where: * `start`: Optional. The starting index of the substring (inclusive). If not specified, the default value is 0. * `end`: Optional. The ending index of the substring (exclusive). If not specified, the

default value is the length of the string. * `step`: Optional. The step size for selecting characters. If not specified, the default value is 1.

[42]:
```
text = "Python is a programming language"
text[7:23]  # all characters between indexes 7 (inclusive) and 23 (exclusive);␣
 ↪step is omitted
```

[42]:
```
'is a programming'
```

[43]:
```
text[7:23:2]  # same as above, using step=2
```

[43]:
```
'i  rgamn'
```

[44]:
```
text[-8:]  # negative values can also be used in slicing
```

[44]:
```
'language'
```

[45]:
```
text[:6]  # first 6 characters
```

[45]:
```
'Python'
```

Getting the length of a string To retrieve the length of a string, we use built-in function `len`:

[46]:
```
text = "Python is a programming language"
len(text)
```

[46]:
```
32
```

1.10.5 Exercises 2.5.2

1. Store the following strings in different variables and check if any one of them is contained in `"The Zen of Python"`:

   ```
   "zen"
   "Python"
   ""
   ```

2. Following the images below for guidance, build and print your first name initial with ASCII art. You will have to use string concatenation and multiplication.

3. Given the string `"abcdefghijklmn"` print the following:

   - the third character of this string.
   - the second to last character of this string.
   - the first five characters of this string.
   - all but the last two characters of this string.
   - all the characters of this string with even indices (remember indexing starts at 0, so the characters are displayed starting with the first).
   - all the characters of this string with odd indices (i.e. starting with the second character in the string).
   - all the characters of the string in reverse order.

- every second character of the string in reverse order, starting from the last one.

1.10.6 String interpolation

F-strings, introduced in Python 3.6, offer a powerful and concise way to format strings with variables and expressions.

Syntax F-strings are prefixed with an `f` before the opening quote. You can embed Python expressions and variables directly within curly braces `{}` inside the string.

Variable Substitution F-strings allow you to easily include variable values within a string.

[47]:
```
name = "Alice"
age = 30
print(f"My name is {name} and I am {age} years old.")
```

```
My name is Alice and I am 30 years old.
```

Expression Evaluation Arbitrary expressions within the curly braces will be evaluated.

[48]:
```
a = 5
b = 3
print(f"The sum of {a} and {b} is {a + b}.")
```

```
The sum of 5 and 3 is 8.
```

Formatting Options F-strings support various formatting options for controlling the appearance of values. The format specifier will be added inside the curly braces, separated by a colon from the variable or expression: `{variable_or_expression:format_specifier}`. A complete guide to format specifiers can be found here. The most common use of format specifiers is for rounding float values to a fixed precision (`.<precision>f`).

[49]:
```
pi = 3.14159
print(f"Value of Pi: {pi:.2f}")
```

```
Value of Pi: 3.14
```

1.10.7 Exercises 2.5.3

1. Given the following variables

```
first_name = 'Mike'
last_name = 'Clarkson'
accounts = 2
balance = 128.5532
```

print the following message using f-strings:

M. Clarkson has 2 bank accounts with a total balance of 128.55$.

1.11 2.6 Console input and output

1.11.1 2.6.1. The `print` function

The `print` function in Python serves as one of the fundamental tools for displaying information to users. It allows you to output text and variables to the console or other output streams. We have already used `print` function to display the results of different operations, but let's see more use cases.

The most basic use case, is calling `print` with one argument (be it a literal or a variable):

[50]:
```
my_int = 10
print(my_int)
```

```
10
```

But `print` (and functions in general) can also be called with an expression as argument. The expression will be computed first, and then its result will be sent to `print`:

[51]:
```
print(my_int * 2 + my_int // 3)
```

```
23
```

`print` can receive multiple arguments when called. The values will be separated by a space when displayed:

[52]:
```
print(my_int, 3.4, "hello")  # calling print with 3 arguments
```

```
10 3.4 hello
```

The default separator can be replaced by a custom one by specifying the keyword argument `sep`:

[53]:
```
print(my_int, 3.4, "hello", sep=", ")
```

```
10, 3.4, hello
```

At the end of each line displayed, `print` adds a newline. Notice how each print output is displayed on a separate line:

[54]:
```
print("hello")
print("hello there")
```

```
hello
hello there
```

This behavior can be overwritten, by specifying the keyword argument `end`:

[63]:
```
print("hello", end=" ")
print("hello there")
```

```
hello hello there
```

1.11.2 Exercises 2.6.1

1. Print the values of the following variables (using a `print` call for each variable):

```
greeting1 = "Hello!"
greeting2 = "Hi!"
question = "How are you?"
answer = "Fine."
```

2. Now do the same thing, but using a single `print` call to display them all on separate lines.

1.11.3 2.6.2. The `input` function

The `input()` function in Python is a powerful tool for getting user input during program execution. It allows you to prompt the user for information and then store their response as a string, which you can manipulate, process, or display as needed within your program.

Using `input()` is straightforward: simply call the function and optionally provide a prompt message as an argument. The program will pause execution, display the prompt message (if provided), and wait for the user to input text followed by pressing the Enter key.

[56]:
```
name = input("Enter your name: ")
```

```
Enter your name:  Iulia
```

[57]:
```
name
```

[57]:
```
'Iulia'
```

Anything the user enters will be saved as a string. If you need variables of a different type, you will have to convert the input to the desired type:

[58]:
```
age = input("Enter your age: ")
```

```
Enter your age:  10
```

[59]:
```
age
```

[59]:
```
'10'
```

[60]:
```
age = int(age)  # converting the string to integer
```

Now we can use the integer age in computations:

[61]:
```
print(f"{name} was born in {2024 - age}.")
```

```
Iulia was born in 2014.
```

1.11.4 Exercises 2.6.2

1. Write a program that will compute the net price and VAT for a product. First, ask the user what product they bought. Then, ask them how much they paid for that product. Considering VAT is 19% of the price, compute the net price of the product and the VAT value. Then, display the following message: > You bought a <product_name> for which you paid <net_price> + <vat_value> VAT.