Python for European Immigrants: Breaking Barriers to Education

Co-funded by the European Union

VOICES OF THE WORLD

Ingenious Knowledge

Table of Contents

Introduction

Welcome to the world of the programming language Python! This book aims to help you learn Python, regardless of your experience. We will explore Python in a step-by-step manner. This book will provide you with the necessary tools to start your journey in Python - even if you have never written a line of code before! We will provide you with a detailed explanation of all the steps necessary to correctly use this powerful programming language, from the choice of software to use to multiple practical examples that will help you master Python and programming as a whole.

Chapter 1 - What is Python

Python was created in 1991 by Guido Van Rossum, and it was designed to be simple, readable, and powerful. Over the years, Python has grown to become one of the most popular languages in the world. Industry reports in 2025 show that 48% of developers use Python regularly, which makes it the **second most popular language globally**.

Why is Python so popular?

- **Beginner-friendly: The syntax (rules of the language) is clear and easy to read.**
- **Versatile:** Python has a wide range of applications, including web development, data science, artificial intelligence, automation, and more.
- **Trusted by big names:** It's used by companies like Google, NASA, and Netflix to power some of their tech innovations.
- **Supportive community:** With millions of users worldwide, it's easy to find tutorials, forums, and libraries to help you learn and build your projects.

Python was built to be used not only by professional programmers but also by scientists, mathematicians, and other people that are not that familiar with programming. This design philosophy makes Python simple, readable, and approachable, allowing experts in different fields to focus on solving their problems without the need to master complex programming concepts first.

Whether you want to build a website, analyze data, create a video game, or create an app, Python gives you the tools to easily bring your ideas to life, and that's why it's one of the best first programming languages you can learn.

Chapter 2 - IDE Introduction

Before we can write Python code, we need a place to type and run it. This place is called an "IDE" (Integrated Development Environment). An IDE is a tool that helps you write, test, and run code. There are multiple IDEs available. We will display our code examples in a universal way, so you can use the IDE of your choice.

We recommend you to use Replit to get started in an easy way. It is free, browser-based (no installation needed), and beginner-friendly. Also, it allows you to directly execute your code by just clicking a "Run" button. This makes it a great tool for us to use in our case.

To get you started on Replit you'll need to follow the next steps:

Step 1: Create an Account

Go to replit.com and create a free account if you don't have one yet.

Step 2: Create a New Project

Once you're logged in, click the "**Create**" button to start a new project as shown on **Image 1**.

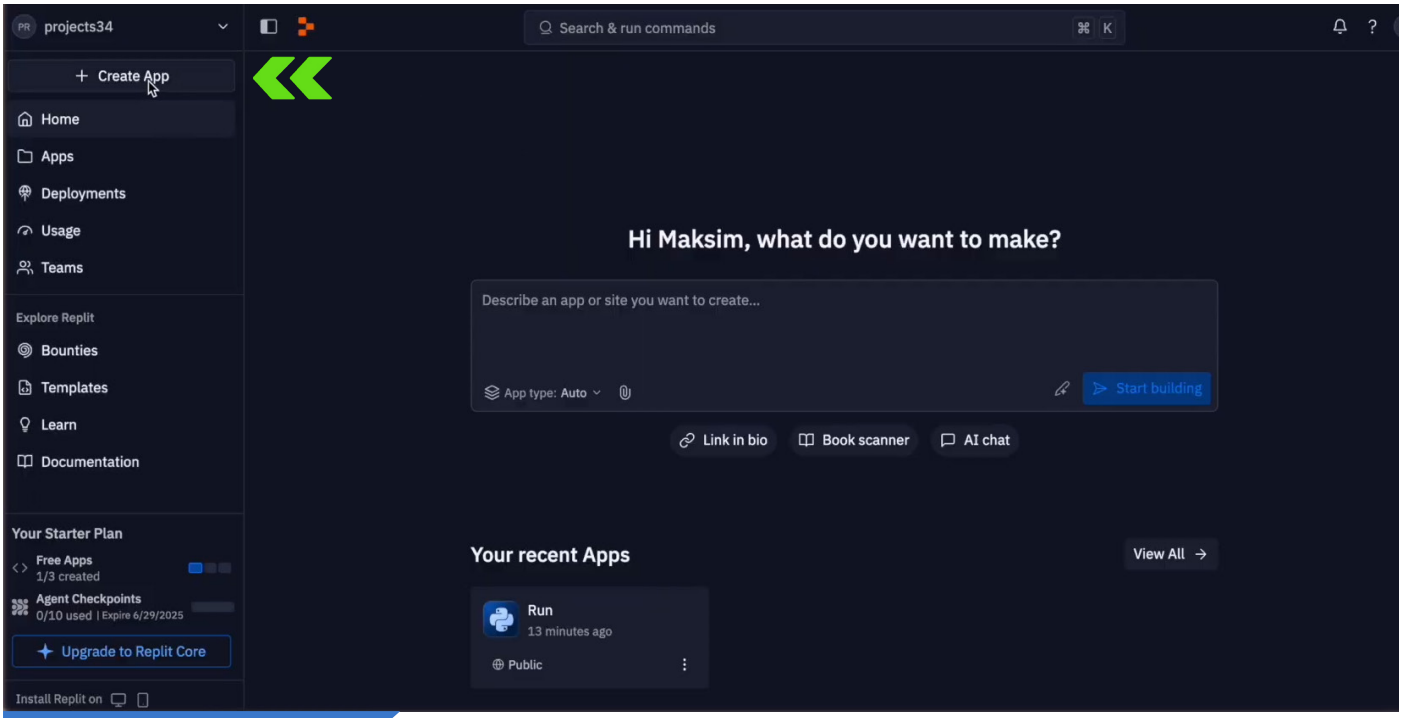

Image 1 - Create project

Then, click on “Choose a Template”, choose Python as your language, give your project a name, and finally click “**Create App**” as shown on **Image 2**.

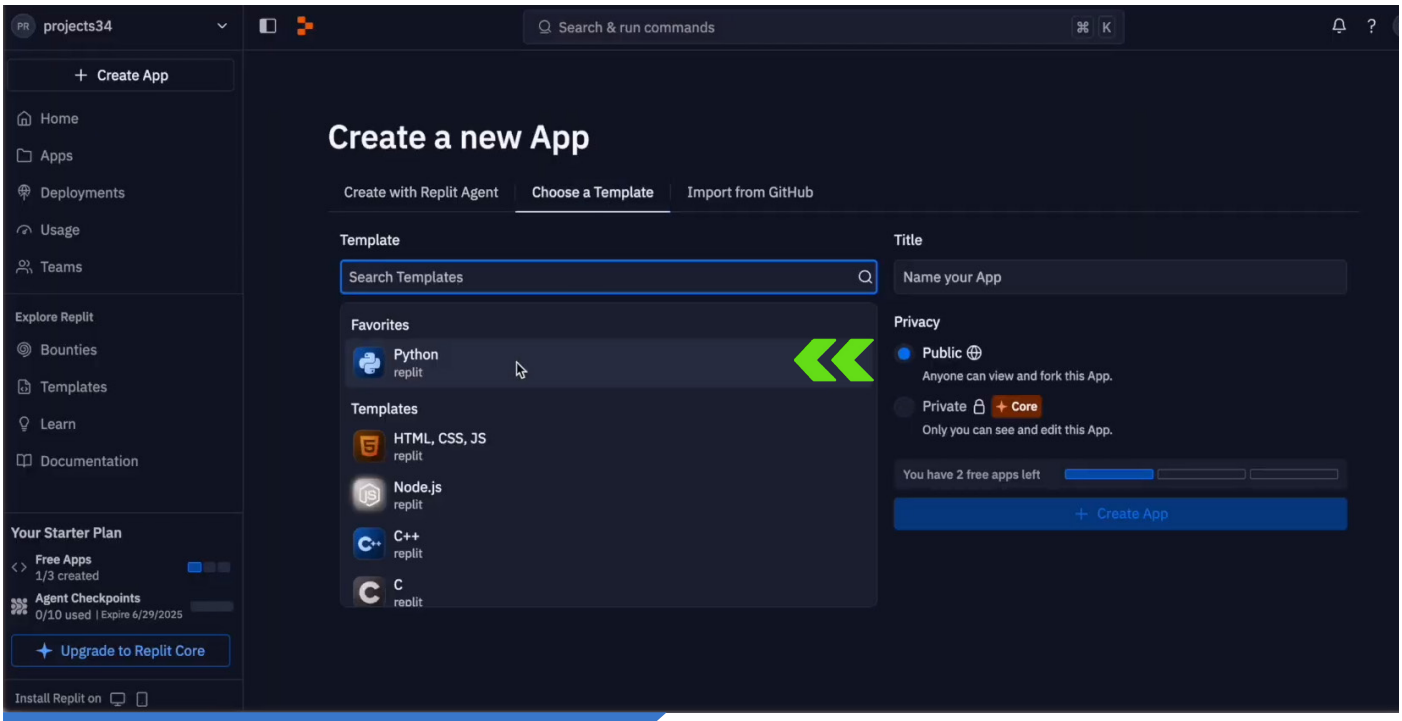

Image 2 - Choose a Template and Create App

Step 3: Explore the Interface

Now, close the "**Assistant**" tab and hit "**Run**" to show the "**console**" tab as shown on **images 3 and 4**.

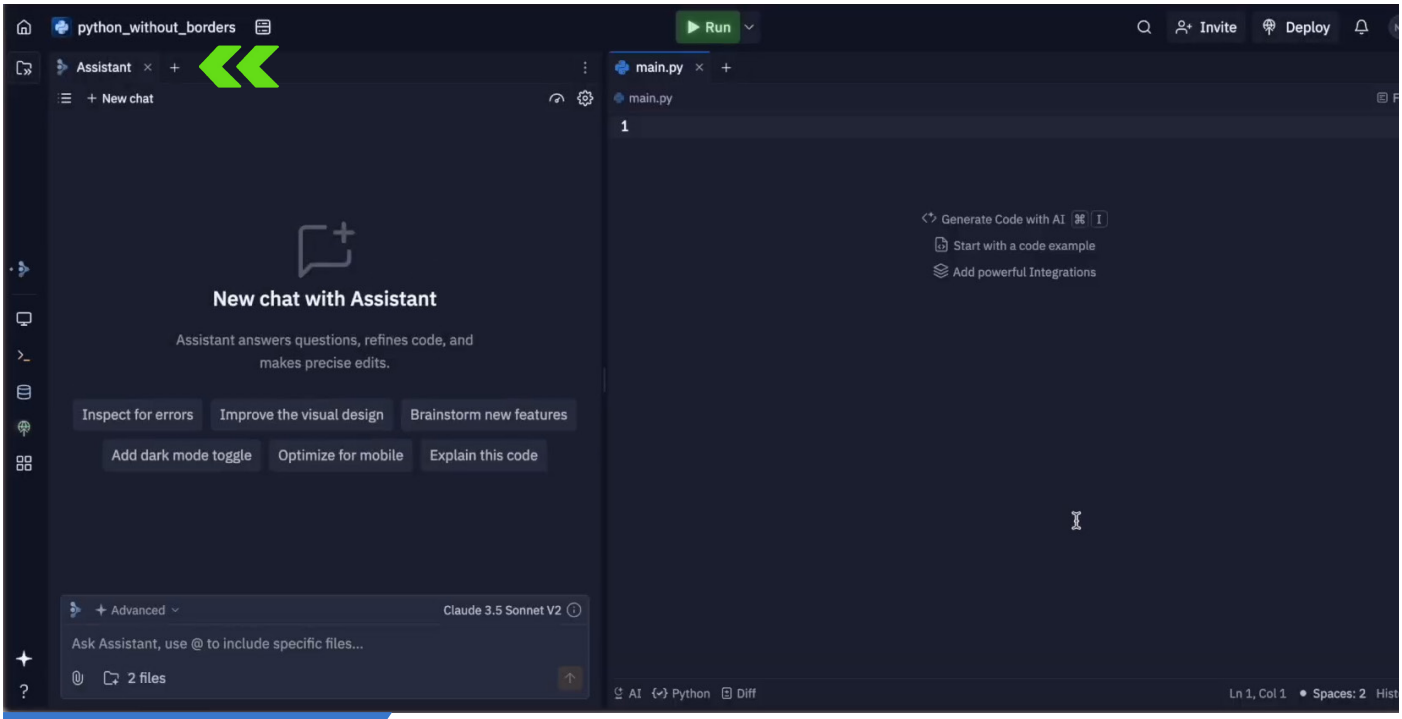

Image 3 - Assistant tab

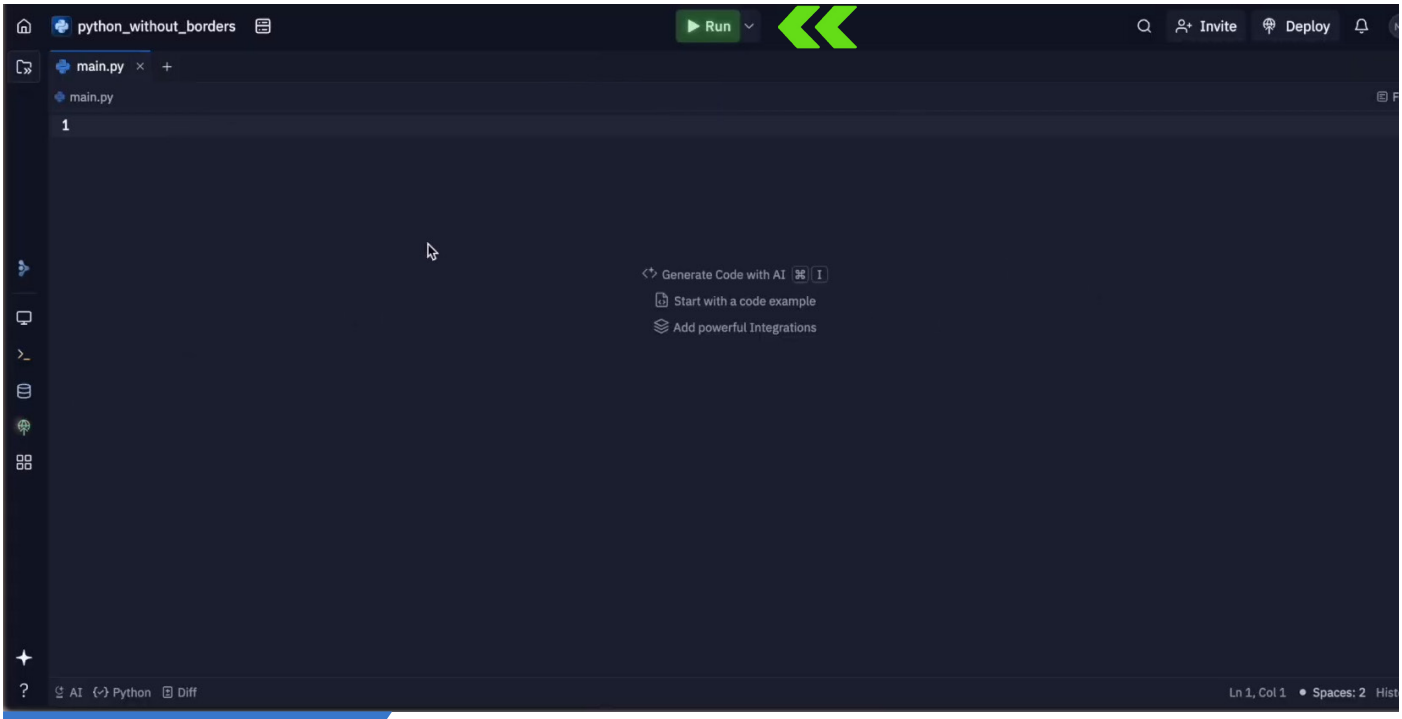

Image 4 - Run button

You will now see two main areas. The left side is where you write your program. Here you can enter the code you want the computer to execute. On the right side you see the console, where your program will run and show the output.

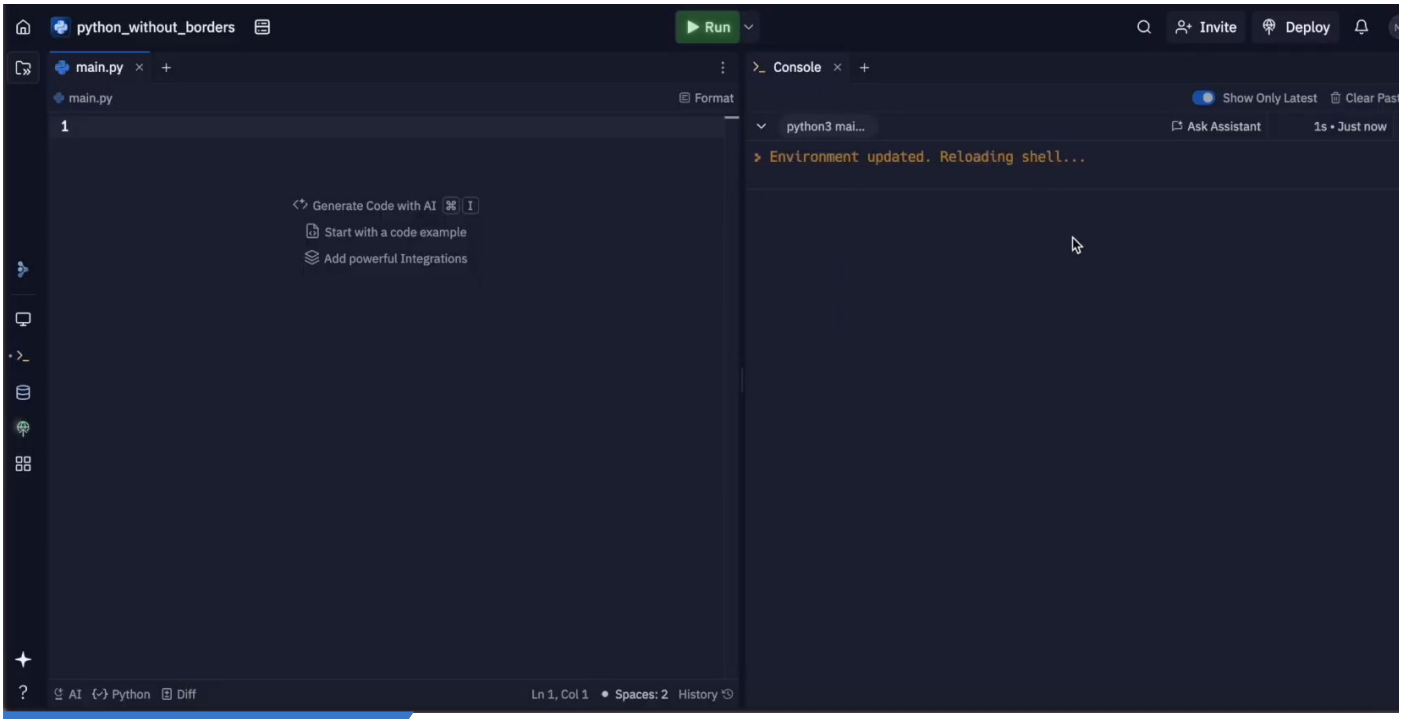

Image 5 - Work environment

Whenever you write code, type it on the left side. When your code is ready, click the "**Run**" button at the top of the screen. Replit will execute your code and show the results on the right.

Chapter 3 - The Print function

Okay! Now that you are ready to code we will start with a very simple program. Every programmer remembers their first program. We will start by letting the computer output a text Hello world!. In Python, this is usually done by using the print() function. The print() function tells the computer to display text or information on the console.

To let the computer know that we want it to output Hello world!, we can write:

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

After executing the program, we get the following output:

```
1 Hello world!
```

The word **print** tells your computer to output something to the console. The parentheses behind the print command contain whatever you want to show on the console. In this case we want to show the words **Hello world!**. The quotation marks indicate that what we want the computer to print is a text. They will not be shown in the console when the program is executed.

We can replace "**Hello world**" with any content we want. For example, if we would write

```
1 print("Something Else")
```

the console would show:

```
1 Something Else
```

Try it yourself!

- Use the **print()** function in your own Python code to make your name appear on the screen.

Solution:

```
1 print("Max")
```

Output:

```
1 Max
```

Chapter 4 - String Variables

Now that you know how to display text, let's take things a step further. Instead of writing text directly inside the **print()** function, we can store it in a **variable**.

A **variable** is like a container that holds information. In the case of a **string variable**, we can store **text** inside a **variable** and then use it later.

In Python, you can create a **string variable** by assigning the text to a name using the "**=**" operator.

```
1 name = "Jenny"
2 print(name)
```

When we run the program, the output is:

```
1 Jenny
```

In this case we write **name = "Jenny"** and then **print(name)**. Please note that in the print command we are not using any quotation marks this time. The reason for this is that we are referencing the variable **name** this time - meaning that we print whatever the value of the variable **name** is just in this specific moment.

Chapter 5 - Concatenation

In the last chapter, you learned how to store text in variables. Now you will learn how to combine strings with each other. This is where **concatenation** comes in. **Concatenation** might sound complicated, but it simply means **joining strings together**.

In Python, to do that, you just need to use the "**+**" operator. Here is an example:

```
1 first_name = "Jenny"
2 last_name = "Smith"
3 full_name = first_name + " " + last_name
4 print(full_name)
```

Output:

```
1 Jenny Smith
```

When the code is executed, Python prints **Jenny Smith** in the console. The "**+**" operator joined the strings together. Notice the **" " (space)** in line 3 keeps the words separated - without it the output would be **JennySmith**, all stuck together. Adding a space in quotation marks keeps the words separate.

Concatenation also works with text variables. For example, I could write:

```
1 name = "Max"
2 print("Hello, " + name + ", I hope you're having a great
      day!")
```

When running the project, the output is now:

```
1 Hello, Max, I hope you're having a great day!
```

Try it yourself!

- Define a variable for a person's name, then use concatenation to print a sentence like this: "My name is **Sarah**".

Solution:

```
1 name = "Max"
2 print("My name is " + name)
```

Output:

```
1 My name is Max
```

Chapter 6 - Integer Variables

So far, we have covered **strings**, which can contain texts as their value. Now you will learn about variables that work with whole numbers. They are called integer **variables**. **Integers** are whole numbers, and they're very useful for math, counting, and more.

In Python, you can create an integer variable by assigning a number to a variable using the "**=**" operator. Let us try out the following code:

```
1 age = 25
2 print(age)
```

Output:

```
1 25
```

In this example the variable **age** holds the number 25. Unlike **strings, integers** do **not** have **quotation marks**. You can replace 25 with any whole number you like. The big difference is that you can do math with these kinds of variables. For example, we could write:

```
1 age = 13
2 year = 2025
3 birth_year = year - age
4 print(birth_year)
```

Output:

```
1 2012
```

We calculate the birth year of someone by doing math using variables. We assign 2025 to the year variable, and 13 to the **age** variable. Then, we create another variable **birth_year** and assign the difference of **year** and **age** to it.

Try it yourself!

- Create an integer variable called value_1 and use print() to display it. Then, define one more integer variable called value_2 and display the result of different operations, such as multiplication, addition, subtraction, and division.

Solution:

```
value_1 = 10
print(value_1)
value_2 = 5

print(value_1 + value_2
print(value_1 - value_2
print(value_1 * value_2
print(value_1 / value_2
```

Output:

```
10
15
5
50
2.0
```

Chapter 7 - Float Variables

We already know how to create variables that hold whole numbers or integers, as we call them. Let us now look at the next variable type: Floats. Floats are numbers with decimals. They are useful when you need to work with prices, measurements, percentages and more.

You can create a float variable by assigning a decimal number. For example, we can write:

```
net_price = 19.99
```

The variable **net_price** becomes a **float variable** and holds the decimal number 19.99. Unlike whole numbers (**integers**), **floats** have a decimal point.

Keep in mind: The decimal point can not be a comma, as Python will not interpret the value correctly then. It has to be a dot.

You can also do math with **float variables**. For example, we can add a tax to our net_price variable and print the total price:

```
net_price = 19.99
tax = 0.21
total_price = net_price + (net_price * tax)
print(total_price)
```

Output:

```
24.1879
```

Try it yourself!

- Create two variables that hold different prices and a third variable that holds their sum. Then, calculate the tax (0.21) of the sum and use print() to display it.

Solution:

```
price_1 = 10.50
price_2 = 6.99
sum = price_1 + price_2
tax = sum*0.21
print(tax)
```

Output:

```
3.6729000000000003
```

Chapter 8 - Naming Variables

Now that we have worked with multiple variables types already, we will take a closer look at the names we choose for them. There are several good practices that you should follow to make your code more readable.

First of all, a variable name should be indicative of what kind of value this variable holds. For instance, if I create a variable that holds the price of a product, it would make a lot of sense to name the variable "price" and not something abstract like "x". This will make it a lot easier for you or even another person to understand your code later.

Another good practice is to start the name of your variable with a lower case letter. Upper case letters have a different meaning in programming. You also want to avoid starting with a number or a special character.

A variable name cannot contain a space. If you use a space, Python will interpret the words you typed as two different variables. So if your variable name consists of more than one word, you need to write these words together.

Finally, there are several ways to make variable names better readable when they consist of several words. For instance, if you call your variable **netPrice**, you would capitalize the **P**. This notation is called **camel case** because the capital letters give the variable name humps like a camel. You can also use **snake case**, which uses an underscore "_" between words. In the net price example, it would look like this: **net_price**.

Examples of valid variable names:

- name
- last_name
- Name (but not recommended by us)
- name_1

Examples of invalid variable names:

- 1name - cannot start with a number.
- first name - spaces are not allowed.
- last-name - the minus sign "-" is not allowed because Python thinks it's a subtraction.
- @age - special characters apart from "_" are not allowed.

Chapter 9 - Booleans

Another important variable type is the **boolean**. **Booleans** store **True** or **False** values, and they are useful for decision-making in programs. In Python, we can create a Boolean variable by assigning **True** or **False** to a name using the "**=**" operator, for example:

```
1 snowing = True
```

The variable **snowing** now holds the **boolean** value **True**. Boolean values are written without quotes, so we will need to write **True** or **False**, not "**True**" or "**False**".

Booleans are essential when you use conditions, but we will get to that later.

```
1 snowing = True
2 print(snowing)
```

Output:

```
1 True
```

Chapter 10 - User Input

So far, our programs have only displayed information - but what if we want to make them more interactive? That's where the **input()** function comes in. It allows users to enter data, making programs interactive and fun.

The **input()** function is used to get text input from the user.

```
name = input("What is your name?
print("Hello " + name + "!")
```

Output:

```
What is your name? Max
Hello Max!
```

As you can see in this example, the code causes our program to ask the user in the console to write something to be assigned to the **name** variable, which will then be printed along with the word **Hello**. The word **input** tells Python to wait for user input. The text inside the parentheses, "What is your name?", is a prompt that guides the user on what to enter.

You can also use **input** to get numbers, for example:

```
age = input("How old are you?
```

However, remember that the **input** function always treats any user input as a **string**. When you want a number, you will have to convert **the user input from a string to a number**. This process will be shown in the next chapter.

Try it yourself!

> Use the input() function to ask the user for their full name, store the input in a variable and then print it.

Solution:

```
full_name = input("What is your full name?")
print(full_name)
```

Output:

```
What is your full name? Sam
Sam
```

Chapter 11 - Typecasting

We will now learn another helpful concept in Python called **typecasting**. It sounds fancy, but it just means changing one type of data to another. Sometimes, you need to convert data to a different type to ensure that your program works correctly. This code snippet shows a pertinent example of why we need **typecasting**.

```
age = 20
print("I am " + age + " years old.")
```

Output:

```
Traceback (most recent call last):
  File "/home/runner/workspace/main.py", line 2, in <module
    >
    print("I am " + age + " years old.")
          ~~~~~~~~^~~~
TypeError: can only concatenate str (not "int") to str
```

We are getting an error, because we are trying to add a number to a string. Python does not allow that. This is when **typecasting** comes in handy. You can convert the number into a string using **str()**, for example:

```
age = 20
print("I am " + str(age) + " years old.")
```

Python turns the number 20 into the string "20", and allows it to be joined to the rest of the string.

You can also **typecast** in other ways. For example, we can change a string into a number by using **int()** or **float()**, as shown in the next example. There, the **string "5"** becomes the **number 5**, so that Python is able to do the math: 5 + 10 = 15. Typecasting is a concept that you will use regularly while programming with Python.

```
number = "5"
result = int(number) + 10
print(result)
```

Output:

```
15
```

Try it yourself!

Ask the user to input a number, multiply it by 7, and then print the result.

Solution:

```
number = input("Enter a number please: ")
number = int(number)
print(number * 7)
```

Output:

```
Enter a number please: 5
35
```

Chapter 12 - Conditions (if & else)

One of the most important concepts in Python is the condition. It allows your program to make decisions. Instead of running the same code, your program can check a condition and choose which code to run.

```
age = 18
if age >= 18:
  print("You are an adult.")
else:
  print("You are a minor.")
```

Output:

```
You are an adult.
```

As you can see in the code snippet above, we are defining the variable **age** with the value 18 and. Then we check if it is greater than or equal to 18 by using the "**>**" and "**=**" operators. If it is greater than or equal to 18, print("You are an adult.") will be executed. If the age variable is less than 18, **print("You are a minor.")** will be executed.

The word if starts the condition. In our example, the condition is age >= 18. Don't forget the colon (:) after the condition - it tells Python that the condition ends here and that the code that needs to be executed follows. If the condition is **True**, Python runs the indented code underneath. If it's **False**, Python moves to the **else** block (if it exists) and runs that code instead.

If you only want to run a code block if a condition is **true**, you can also just use if without the **else** as shown in the code snippet below.

```
age = 17

if age >= 18:
  print("You are an adult.")
```

Output:

```

```

Another important reminder is that you need to indent the block of code that should only be run if the **if** statement is **true**. Without the indentation, Python would be unable to understand when to run the code block, and it would always run it. However, we will talk more about indentation later.

In our previous example, we have used the **greater or equal** comparison (**>=**). We can also check if a value is **less (<), greater (>) , less or equal (<=)**, or, when we check if a value is equal, we use two equal signs like this "**==**".

We can also use the "**==**" comparison for strings. In the next code snippet, we have an example of a password checker using the "**==**" comparison. There, we check if the string that the user writes is "secret", if it is, we print "You may pass." Otherwise, we print "Access denied!".

```
password = input("Password: ")

if password == "secret":
  print("You may pass.")
else:
  print("Access denied!")
```

Output:

```
Password: secret
You may pass.
```

Try it yourself!

- Use an **if/else** statement in your own Python code to check if the user guesses a number correctly. If the user writes the correct number, the program should print "You are right!". Otherwise, it should print, "Wrong number."

Solution:

```
number = int(input("Guess a number please: "))
secret_number = 34124

if number == secret_number:
  print("You are right!")
else:
  print("Wrong number.")
```

Output:

```
Guess a number please: 34124
You are right!
```

Chapter 13 - Indentation

An important feature of Python that makes it different from other languages is **indentation**. **Indentation** plays a big role in how Python understands your code.

Indentation means adding spaces and tabs at the beginning of a line. This is not just for making the code look better, it actually tells Python what belongs together. Let's take a look at the example below.

```
if True:
  print("This line is indented!")
print("This line is not.")
```

Output:

```
This line is indented!
This line is not.
```

As you can see in the output, Python prints both lines. The first **print** is indented, so Python knows that it is part of the **if** statement. The second **print** is not indented, so it runs after the if block. If we change the condition from **True** to **False**, only the **print** that's not inside the **if** statement is executed as you can see on the next example.

```
if False:
  print("This line is indented!")
print("This line is not.")
```

Output:

```
This line is not.
```

If you forget to indent when you need to, Python will give you an error "**IndentationError**" That's because Python is expecting the next line to be indented. This is how the error would look like:

```
if False:
print("oops!")
```

Output:

```
File "/home/runner/workspace/main.py", line 2
print("oops!")
^
IndentationError: expected an indented block after 'if'
    statement on line 1
```

A helpful rule is to indent every time you use a colon (:). For example, after the "**if**" statement, the next line should be indented. Python usually uses **4 spaces** for each indentation level. You can also use the Tab key, which is the usual way programmers do it.

Try it yourself!

- Write a program with a condition. If it is true, print one text. If it is false, print another text. Print a third text after the condition.

Solution:

```
if True:
  print("This should only be printed if the condition is
      True.")
else:
  print("This should not be printed if the condition is
      True")

print("This should always be printed.")
```

Output:

```
This should only be printed if the condition is True.
This should always be printed.
```

Chapter 14 - Elif

You already learned how to use if and else, but there is another keyword: elif. Elif is short for else if, and it helps your program choose between multiple options. In Python, elif is used when you want to check more than one condition after your initial if. Here is an example:

```
temperature = 21
if temperature > 30:
  print("It is hot outside!")
elif temperature > 20:
  print("It is a nice day!")
else:
  print("You might need a jacket.")
```

Output:

```
It is a nice day!
```

In the code snippet Python checks the first condition. If the condition is false, it moves to the **elif**. If the elif is true, it runs that block. Otherwise, it goes to the **else** block.

The word **elif** is a way to say: "If the first thing isn't true, then check this other thing." You can have as many **elif** statements as you need, and they make your code easier to read and organize.

Try it yourself!

- Create a program that asks the user about the weather and gives clothing advice. Use one **if**, multiple **elif**, and one **else** statement to handle different types of weather, such as **sunny, rainy, cold,** and **snowy**. For any other input, display a default message. In short, print different messages depending on the input entered.

```
weather = input("What is the weather like today? (sunny, rainy, cold, snowy): ")
if weather == "sunny":
  print("Wear sunglasses")
elif weather == "rainy":
  print("Take an umbrella")
elif weather == "cold":
  print("Wear a warm jacket")
elif weather == "snowy":
  print("Put on boots and gloves")
else:
  print("I don't know that weather, just dress comfortably")
```

Output:

```
What is the weather like today? (sunny, rainy, cold, snowy): sunny
```

Chapter 15 - Comparison operators

We will now take a closer look at comparison operators in Python. They let your program compare values and they are useful for making decisions. In Python, comparison operators are used in conditions to check if things are **equal, greater, less**, and more. Check the code below.

```
a = 10
b = 5
print(a > b)
```

Output:

```
True
```

When the code is executed, Python checks if **a** is greater than **b**. Since 10 (**a**) is greater than 5 (**b**), it prints **True.**

Here are some of the most common comparison operators:

== means "**equal to**"

!= means "**not equal to**"

> means "**greater than**"

< means "**less than**"

>= means "**greater than or equal to**"

<= means "**less than or equal to**"

You'll often use these in if statements, as you can see on the code below.

```
age = 16
if age >= 18:
  print("You can vote")
else:
  print("You are too young to vote")
```

Output:

```
You are too young to vote
```

Try it yourself!

Write a program that asks the user for their age and prints messages based on their age using the comparison operators. If the user is **18 or older**, print "**You are an adult.**", if the user is **between 13 and 17**, print "**You are a teenager.**", if the user is **12 or younger**, print "**You are a child.**" and finally, if the user's age is **exactly 21**, print "**Happy 21st birthday!**".

```
age = int(input("Enter your age: "))
if age >= 18:
  print("You are an adult.")
elif age >= 13:
  print("You are a teenager.")
else:
  print("You are a child.")

if age == 21:
  print("Happy 21st birthday!")
```

Output:

```
Enter your age: 13
You are a teenager.
```

Chapter 16 - The randint function

The **randint** is an important Python function that allows you to add randomness to your code.

In Python, **randint** is used to generate a random integer between two values. But first, since this function is not part of the basic Python functions, you need to import the random module which defines how the function works. To import a module from a different library, we use the **import** word followed by the module we want to import. Now, we might not need to import all the functions of a library, in that case we also use the word **from** followed by the library we want to use, then we write **import** and finally the module from that library as displayed on the example below.

```
from random import randint

number = randint(1, 10)
print(number)
```

Output:

```
1
```

When you call the **randint** function, Python picks a random number between two numbers, in our case, between 1 and 10 including both 1 and 10. The word **randint** stands for "**random integer**", and the two numbers inside the parentheses tell python the range from which you want to select the number from. You can change them to anything you like as long as they are Integer numbers, for example **randint(100,200)**, this will return a random **Integer** between 100 and 200.

Try it yourself!

- Write a program that uses the **randint** function to pick a random number between 1 and 100, then print the result with the message "Your surprise number is **X**!", where **X** is the number.

Solution:

```
from random import randint

number = randint(1, 100)
print("Your surprise number is " +  str(number) + "!")
```

Output:

```
Your surprise number is 94!
```

Solution:

```
number = int(input("Enter a number: "))

if number % 3 == 0:
  print("This number is a multiple of 3.")
else:
  print("This number is not a multiple of 3")
```

Output:

```
Enter a number: 5
This number is not a multiple of 3
```

Chapter 17 - The modulo operator

The task of the modulo operator (%) is very simple, it gives you the remainder after division. It is a very simple concept but with a lot of potential and usage in programming.

```
print(10 % 3)
```

Output:

```
1
```

As we can, **print(10 % 3)** gives the output 1. That is because 10 divided by 3 is 3 with a remainder of 1. The % symbol tells your computer, "Give me what is left over after dividing these two numbers." It is very helpful when you want to check for even or odd numbers, like the next example. In that case, if the remainder is 0, then the number is even, otherwise, it is odd.

```
number = 7

if number % 2 == 0:
  print("Even")
else:
  print("Odd")
```

Output:

```
Odd
```

Try it yourself!

- Create a program that checks if a number entered by the user is a multiple of 3. If it is, print "This number is a multiple of 3", if not, print "This number is not a multiple of 3".

Chapter 18 - While Loop

Loops are a very important part of programming as they allow us to repeat pieces of code until something changes. In Python, a **while** loop keeps running **while** the condition is **True**, as shown below.

```
count = 1

while count <= 5:
  print(count)
  count = count +1
```

Output:

```
1
2
3
4
5
```

Python prints the numbers 1 through 5. That's because the word **while** tells your computer: "Keep looping as long as this condition is **True**." In this case, we keep looping while **count** is **less than or equal to 5**. Each time through the loop, we print the number and then **add 1 to the count variable**. Without changing the **count** variable, the loop would go on forever. Therefore we always need to make sure the loop will eventually stop.

Note that we indented everything that happens inside the loop to help Python understand which part of the code the loop should repeat.

Try it yourself!

- Write a **while** loop that returns the answer to the factorial of 20 and print the result. The Factorial of a number is the product of all positive integers from 1 up to the given number (e.g., 5! = 5 × 4 × 3 × 2 × 1 = 120).

Solution:

```
number = 20
factorial = 1
counter = 1

while counter <= number:
  factorial = factorial * counter #multiply the current
      factorial by counter
  counter = counter + 1 #move to the next number

print("The factorial of 20 is: " + str(factorial))
```

Output:

```
The factorial of 20 is: 2432902008176640000
```

Chapter 19 - Loops inside loops

Loops can also be used inside other loops. This is also known as nested loops. Nested loops let you code actions in a grid-like or layered way. Below, we can see an example of how nested loops are implemented.

```
for i in range(10):
  row = ""
  for j in range(10):
    row = row + "."
  print(row)
```

Output:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

When the code is executed, Python prints pairs of numbers: Every combination of "i" and "j" from 1 to 3. The outer loop runs first. For each iteration, the inner loop runs completely. Imagine it like a clock: For every hour, the minute hand completes a full rotation. You can use nested loops to do things like print patterns, build grids, or work with rows and columns of data.

Notice that we used a different loop, called a for loop. We will get to them in a later chapter.

Try it yourself!

- Use a loop inside a loop to create a square of "." dots with the dimensions of 10 by 10.

Solution:

```
for i in range(10):
  row = ""
  for j in range(10):
    row = row + "."
  print(row)
```

Output:

```
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
```

Chapter 20 - Lists

Lists are very useful when you want to store a collection of items, like a to-do list or a group of names. In Python, a list is defined by a set of values wrapped in square brackets just like the example below.

```
fruits = ["apple", "banana", "cherry"]
print(fruits)
```

Output:

```
['apple', 'banana', 'cherry']
```

Each item in the list is separated by a

comma, and you can add, remove, or access items by their position (**index**). For example, you can access the **apple** string by writing **fruits[0]** as displayed on the code snippet below. Attention: the index always starts at **0**.

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])
```

Output:

```
apple
```

The output is just “apple”, because it is the first item. It is important to keep in mind that in Python the index of lists always starts from 0 and not 1. If you want to access the second item of a list, in this case you would need to write **fruits[1]**, to access the last item of a list it would be **fruits[-1]**. You can save all kinds of things in lists, **Integers, String, Booleans** and many more types of data. Lists are a very good option when you need to store multiple values in one place. Lists also work especially well with loops, as we will see in the next chapter.

Try it yourself!

- Use this list “carBrands = [“Toyota”, “Mercedes”, “Nissan”, “Opel”, “BMW”, “Fiat”]” and try to print “Toyota”, “Nissan”, and then “Fiat” while only using the index values.

Solution:

```
carBrands = ["Toyota", "Mercedes", "Nissan", "Opel", "BMW",
    "Fiat"]

print(carBrands[0]) #Toyota (first item, index 0)
print(carBrands[2]) #Nissan (third item, index 2)
print(carBrands[-1]) #Fiat (sixth item, index 5 or last
    index which is “-1”)
```

Output:

```
Toyota
Nissan
Fiat
```

Chapter 21 - for loops

A for loop is a programming mechanism that allows you to repeat a block of code a specific number of times or iterate through items in a sequence like a list. The for loop is commonly used when you know in advance how many times you want to run the loop, or when you need to go through each element in a list.

```
for i in range(5):
  print(i)
```

Output:

```
0
1
2
3
4
```

When the code above is executed, it prints the numbers 0 through 4. The word for tells your computer to loop through each item. i is the variable that changes each time, and range(5) makes the loop iterate 5 times, giving us the numbers from 0 to 4 (remember that Python always starts counting at 0). You can replace the i with any other variable.

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
  print(fruit)
```

Output:

```
apple
banana
cherry
```

You can also use **for** loops to go through items in a list, as you can see above. It prints each fruit in the list one at a time. **fruits** is a variable of the type **list**.

While loops and **for loops** can often be used interchangeably, however, as a rule of thumb, you should use **for loops** when you know in advance how many iterations the loop will run through.

Try it yourself!

- Use a **for** loop to print all the car brands in this list "carBrands = ["Toyota", "Mercedes", "Nissan", "Opel", "BMW", "Fiat"]".

Solution:

```
carBrands = ["Toyota", "Mercedes", "Nissan", "Opel", "BMW",
    "Fiat"]
for brand in carBrands:
  print(brand)
```

Output:

```
Toyota
Mercedes
Nissan
Opel
BMW
Fiat
```

Chapter 22 - Dictionaries

In the programming world, a **dictionary** is just like a real-world dictionary. But instead of words and definitions, it stores pairs of information, like a person's name and age. They work using keys which are like the words in a real dictionary and **values** which are like the definitions of those words. Each **key** points to a **value**.

```
person = {"name": "Alice", "age": 30}
print(person)
```

Output:

```
{'name': 'Alice', 'age': 30}
```

In the example above, the **person variable** is a dictionary. The **key**, **"name"** is pointing to the **value "Alice"**, and the **key "age"** is pointing to the **value "30"**. The word before the colon is the key, and the word after is the **value**.

To create a dictionary, you use an opening curly brace at the start ({), followed by the pairs of **key** and **values** separated by commas (,), then to end the dictionary, you need to add a closing curly brace (**}**).

```
person = {"name": "Alice", "age": 30}
print(person["name"])
```

Output:

```
Alice
```

The most useful feature of the dictionary is that you can access the **values** by referring to their keys, like the example above, where we are requesting the value of the variable **person** with the **key name**, which results in the output "**Alice**".

Dictionaries are great when you want to organize information clearly and make it easily accessible, like contact info, game scores, or settings.

Try it yourself!

- Use a dictionary to store all your physical features in a variable, for example, eye color, hair color, height, and gender. Then, print your hair color using the dictionary.

Solution:

```
myFeatures = {
"eye_color": "brown",
"hair_color": "black",
"height" : "1.80m",
"gender": "male"
}
print("My hair color is: " + myFeatures["hair_color"])
```

Output:

```
My hair color is: black
```

Chapter 23 - functions

One of the most important building blocks in Python are **functions**. They help you organize your code and make it reusable. A **function** is a block of code that runs when you call it.

To create a function, you start by writing **def** followed by the name of the function and then parentheses and a colon, as displayed below.

```
def greet():
  print("Hello there!")
```

If you execute the code, nothing will happen because the **function** is not being called anywhere. To use it, we need to call it, and the way we do that is by writing the name of the **function** along with the parentheses, as you can see below. When you call the function, all the code inside of it is executed, in this case **print**("**Hello there!**").

```
1 def greet():
2   print("Hello there!")
3
4 greet()
```

Output:

```
1 Hello there!
```

Inside the parentheses you can add inputs called parameters. Parameters are like placeholders or "empty boxes" a **function** uses to receive information. When we are creating a function, we give it **parameters**, which are the names of the variables we are expecting. When we call the **function**, we provide **arguments**, which are the actual values that will be stored in those variables.

```
1 def isEven(number):
2   if (number%2 == 0):
3     print("Your number is even.")
4   else:
5     print("Your number is odd.")
6
7 userInput = int(input("Type a number:  "))
8 isEven(userInput)
```

Output:

```
1 Type a number:  5
2 Your number is odd.
```

In the code above, we are creating a **function** that has the parameter **number** and, when called, it verifies if the number is odd or not, printing different messages based on the result.

Let's break it down a little: lin line **1**, we are defining the function, in line **7**, we request the user to enter a number on the console and then use **typecasting** to convert the **string** to an **integer**. Then, on line **8**, we call the function **isEven** and pass the variable **userInput** as the argument, which means that, when the **function** is executed, the **number** parameter takes the value of **userInput.**

The advantage of this approach is that you can reuse this code block whenever you need to check if a number is odd or even. All you need to do is call the function with a single line:

```
1 isEven(number)
```

Try it yourself!

Create a function called "**welcomeMessage**" that receives a name as a parameter and prints a customized welcoming message with the name entered by the user with the **input** function.

Solution:

```
1 def welcomeMessage(name):
2   print("Welcome to the program, "+ name + "!")
3
4 user_name = input("Enter you name: ")
5
6 welcomeMessage(user_name)
```

Output:

```
1 Enter you name: Max
2 Welcome to the program, Max!
```

Chapter 24 - Multiple function parameters and return

In the previous chapter, we already covered the basics of how functions work. However, there are still more aspects of them, like the return values or working with multiple parameters. Those aspects make functions even more powerful.

So far, in the examples provided, we have only seen functions with no parameters or with 1 parameter. However, you can use many parameters just as long as you separate them with a comma (just like you do with the items of a list).

```python
def add(a, b):
  result = a + b
  return result

additionResult = add(3, 5)

print(additionResult)
```

Output:

```
8
```

On the code above we can see a function with two parameters and a **return**. If we removed line 3, that function would be useless since we are not using the result of the addition. That's why we are using the keyword **return** to create the return value of this function. When executed, the function takes two numbers and returns their sum, **8**. The **return** keyword tells Python to give back a value. That value can then be stored in a variable, used in another calculation, or printed out just like what we are doing in line **5** by assigning the return of the function **add** to the variable **additionResult**.

Without the **return**, the function just runs and finishes. With the **return**, you can keep and reuse what it creates.

Another important detail of the functions is that, if you have a variable inside of them, you cannot access it outside, for example, in the example below, inside the function we are defining the variable **result** as **1**, and outside we have a variable with the same name but with the value **999999**, at the end of the code, we print the variable value. The value printed is the value of the variable that's **outside** the function.

```python
def randomFunction():
  result = 1

result = 999999

print(result)
```

Output:

```
999999
```

Try it yourself!

Write a function that returns the average of 3 numbers, then print the result.

Solution:

```python
def average_of_three(a, b, c):
  return (a + b + c) / 3

num1 = 10
num2 = 20
num3 = 30

result = average_of_three(num1, num2, num3)
print("The average is: " + str(result))
```

Output:

```
The average is: 20.0
```

Co-funded by
the European Union

VOICES
OF THE
WORLD

IngeniousKnowledge