



Quick tutorial for beginners

PYTHON IN SIMPLE WORDS



print()
int(input())
while
range()

Alexandros Kofteros, PhD

Python in simple words

Quick tutorial for beginners

Translation in English: Zoi Karageorgiou

© 2023 Alexandros Kofteros. All rights reserved.
Distributed digitally under a Creative Commons
Licence.



Nicosia, 2023
978-9925-8055-0-1

Images were created with Microsoft Bing



Special Thanks

for their valuable time in reading each version of the guide and continuously sending detailed suggestions and feedback:

Pola Misthou, teacher (Greece)

Vaso Servou, teacher (Greece)

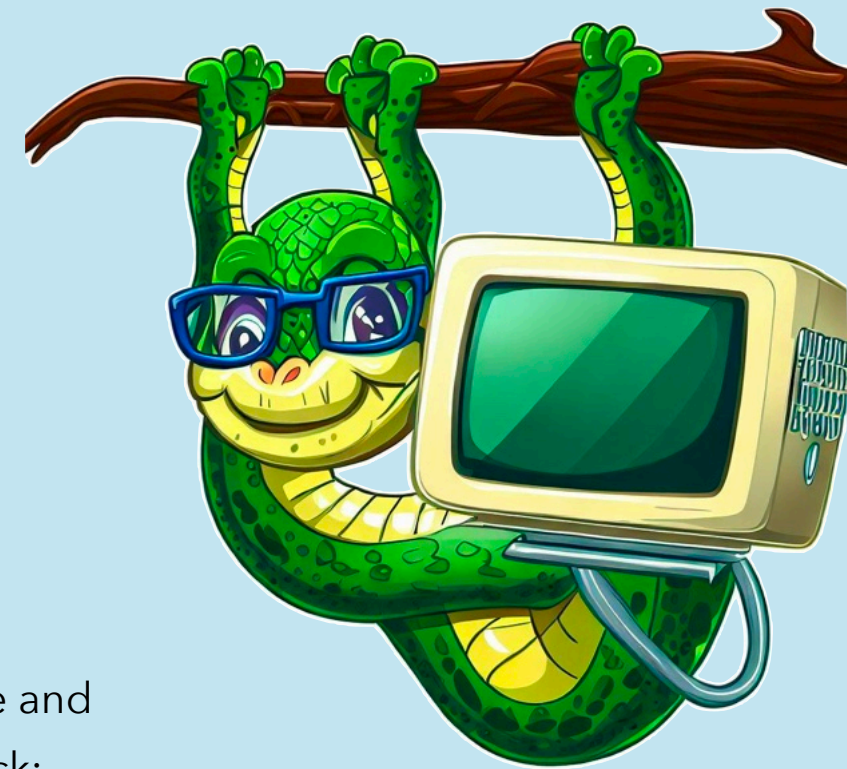
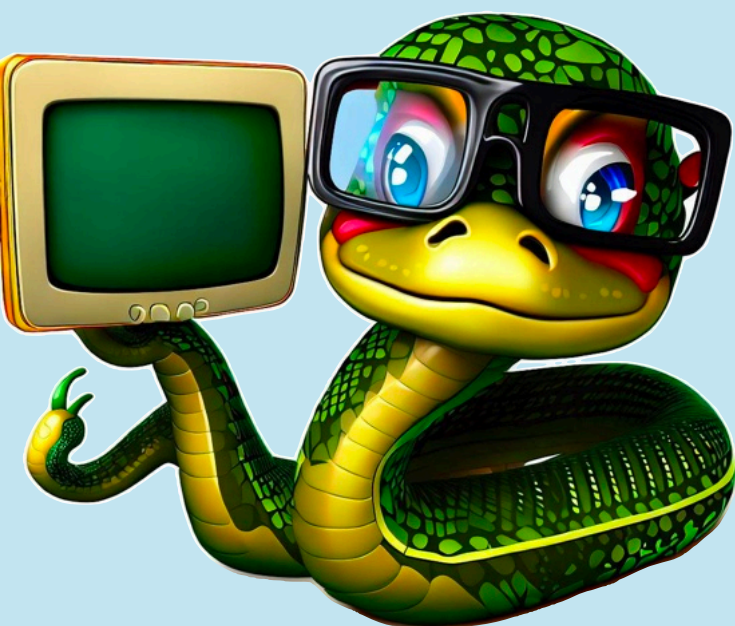
Zoi Karageorgiou (Greece)

Diamanto Georgiou, teacher (Cyprus)

Antonis Foinikarides, IT (Cyprus)

George Kyprianou, educator (Cyprus)

Marios Charalambous, graphic designer (Cyprus)



In this book we are going to...

- acquaint ourselves with the basic features of programming languages
- get to know the history of Python
- learn and use Python commands
- create geometric shapes with Python
- program devices like micro:bit and MeetEdison using commands in Python



About this book

Welcome to our book for Python! This is the first attempt to create a simple introduction guide to this programming language, aimed at teachers and students aged 10+.

This book primarily targets individuals new to programming, embarking on their initial learning journey, and aspiring to acquaint themselves with a reasonably proficient language.

The book is divided into four parts: in the first part we will learn more about programming languages, with an emphasis on Python. In the second, we will get to know basic Python commands, through simple examples. In the third we will cover additional functions of Python, with an emphasis on creating geometric shapes. In the fourth part we will get to know Python programming environments for IoT and educational robotics applications (BBC Micro:bit and MeetEdison).

Welcome!

The guide in front of you (or book, if you prefer) started in July 2021 as an aid (notes) for my son, to learn Python. Along the way, Konstantinos took up C++, which he really likes, while I put the Python book on hold to focus on the Computer Museum's educational program. Two years later, and after going through TOO many changes, the first edition of my book is ready.

The reason for its creation is simple: I wanted to return to the logic of the 1980s, where we opened our computer (Spectrum, AMSTRAD CPC, Commodore 64, etc.) and saw in front of us the BASIC options. For some reason, we all liked to create an infinite loop.

By the same token, I wrote a book aimed at young ages (10+) that attempts to put our fingers back on the keyboard!



Python! Monty Python!



A vibrant, cartoon-style illustration of a computer office. In the foreground, a green worm with large, expressive eyes sits in a green ergonomic chair, looking towards the left. To its right, another green worm stands behind a desk, smiling and holding a telephone receiver. The desk is cluttered with papers, a keyboard, and a mouse. In the background, there are several computer monitors on desks, a yellow desk lamp, and large windows letting in bright sunlight. A yellow cup sits on the floor to the left. The overall scene is bright and cheerful.

PART A': COMPUTER PROGRAMMING

1. Introduction to Programming

```
10 | print ("And what is the use of a book  
20 | without pictures or conversation?")  
50 | #Alice in Wonderland
```


1. What is Python?

Everyone knows that pythons are cute animals. Of course, we would never advise you to approach one, not even as a joke. If you ever happen to be in a country with pythons, it's advisable to steer clear of areas where they inhabit. They don't like strangers, especially people, approaching their nests.

In this book, of course, we will not deal with these nice animals (which, as we said, we won't bother them), but with a **programming language**, particularly widespread and capable of creating complex applications.

Did you know?

The language got its name, not from this lovely animal, but from the legendary **Monty Python**. Oh yes, that is the "secret" behind the name of this language.





What we are going to learn:

In **Chapter 1: "Introduction to Programming"**, we are going to learn:

- What are programming languages?
- How man communicates with computer through programming?
- Which programming environments do we use in education?
- The creation of the Python language.
- How do we download and install Python on our computer?

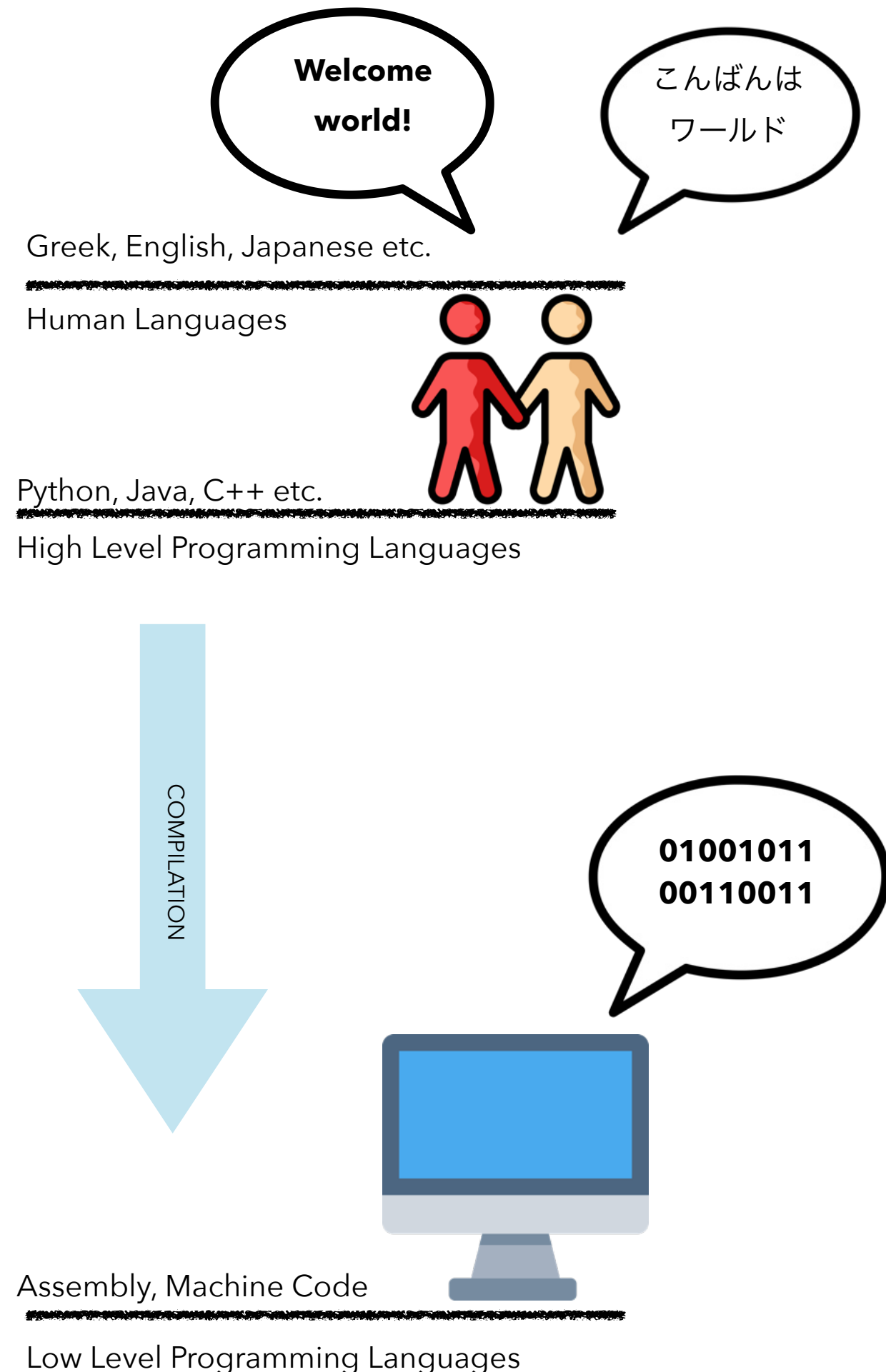


A **program** (or code) is a series of commands that we give, usually via keyboard, to execute one or more instructions.

The first computers were built to do very specific tasks (eg specialized mathematical calculations). These computers were mainly used in the 1940s and 1950s and were programmed in "machine language", a language that makes sense to the computer but not to humans (with the exception of "machine language" connoisseurs).

Programming this way was very complex, and as computers evolved - and gained new capabilities - it was very difficult even for the most skilled programmers.

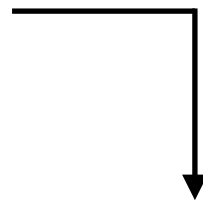
In the 1950s, the first programming languages that were developed were -somewhat- simpler than assemblers, but were still complex. At the same time, programming languages closer to our own began to be developed. Some of the best known of the 1950s were FORTRAN, developed by IBM, and COBOL, developed by Grace Hopper.



In this book we will deal exclusively with Python. But computer programming in schools began in the 1960s with the creation of **LOGO** in 1967 by Seymour Papert, a tireless teacher who especially loved children and education.

With LOGO, which they initially programmed with a ground robot, they were able to create geometric shapes. For example, to create a 90 degree angle, we must give the commands:

```
FORWARD 10  
RIGHT 90  
FORWARD 10
```



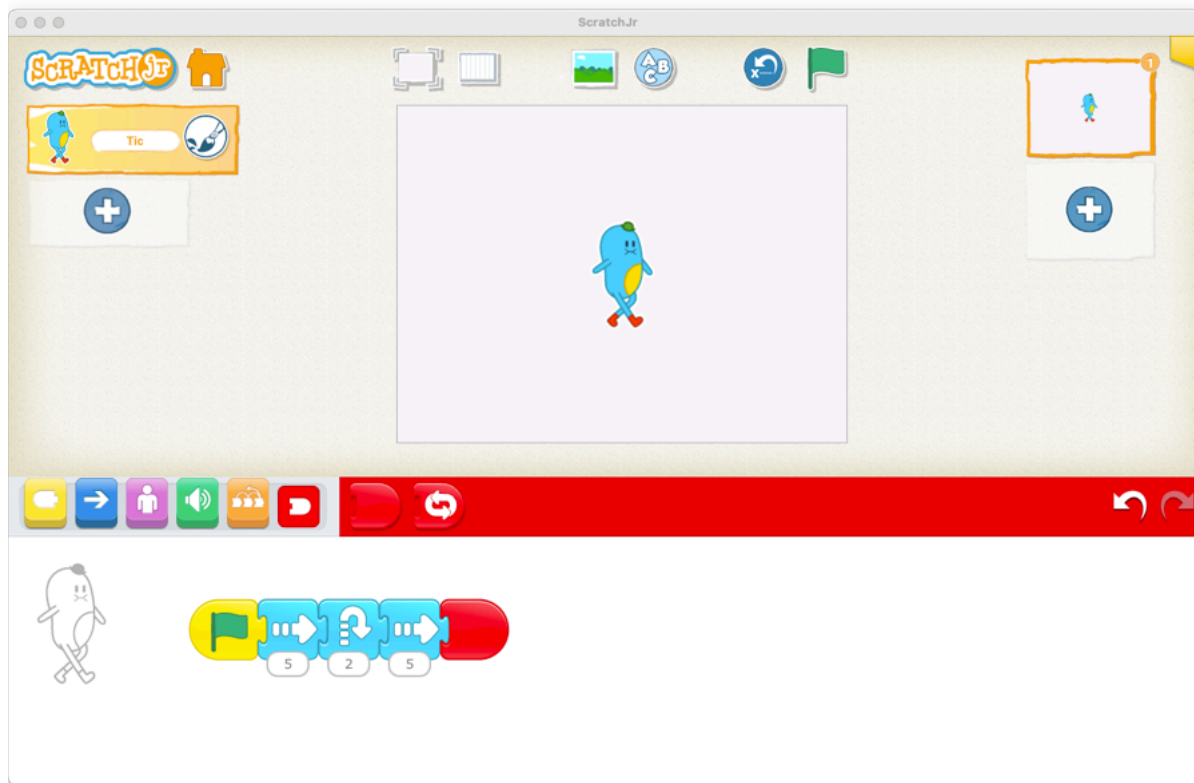
Another language that was used quite a bit in schools, mainly in the United Kingdom, was BASIC (Beginners All-purpose Symbolic Instruction Code).

It is one of the simplest programming languages created, and was mainly used by novice users. In the 1980s, all personal υπολογιστές had integrated in memory (ROM) or in a removable media (tape or disc) a version of BASIC.

In BASIC, to display the Hello World message on the screen, we had to type the following command:

```
10 PRINT "HELLO WORLD"
```

BASIC has now been replaced by other languages or programming environments that offer great convenience to the novice user and more features. However, some software development environments, such as AOZ Studio, are based on BASIC.

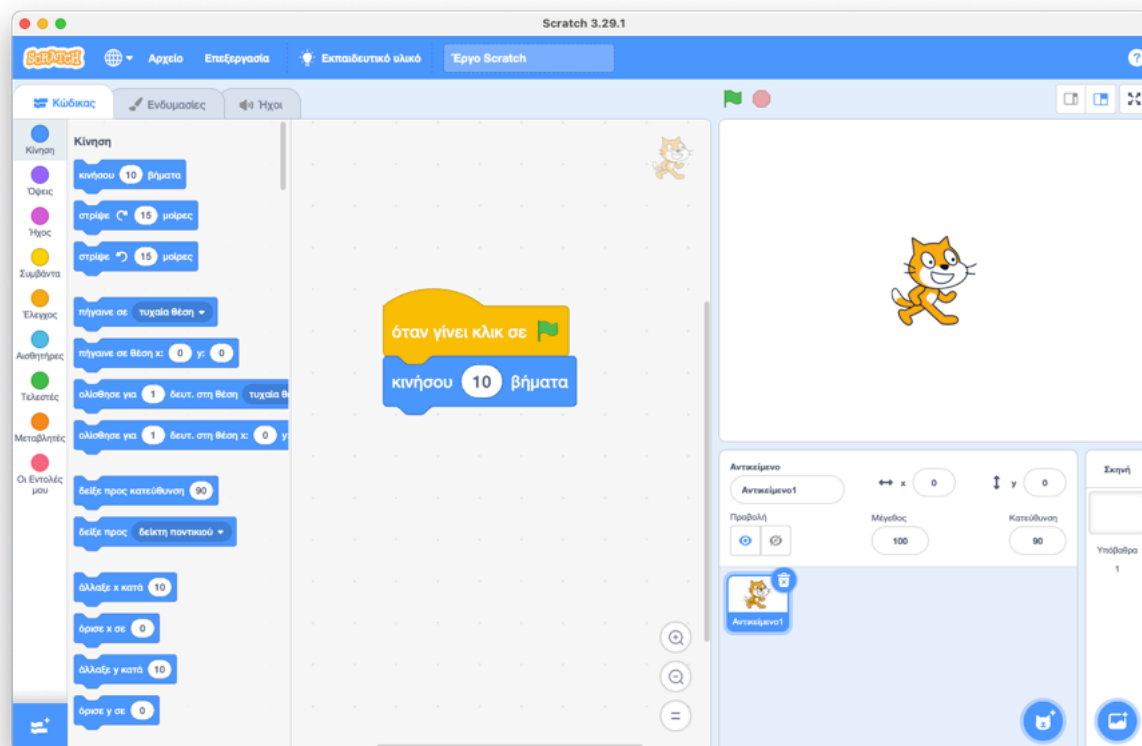


Programming in **LOGO** and **BASIC** at schools, and especially at a young age, encountered a serious problem (there were others, but let's stick to this one...):

the children had to memorize several commands and all of them had to know the English language, as there was no LOGO or BASIC (their commands) in other human languages.

Both of these problems, at least for the small classes (Pre-Primary - 4th Grade) were solved before the end of the 2000s, with the development of the Scratch programming environment, which was based on blocks that any user can connect and create a program. **Scratch Jr** for tablets followed a little later, with an even simpler environment, which easily allows Pre-Primary and First Grade children to learn programming.

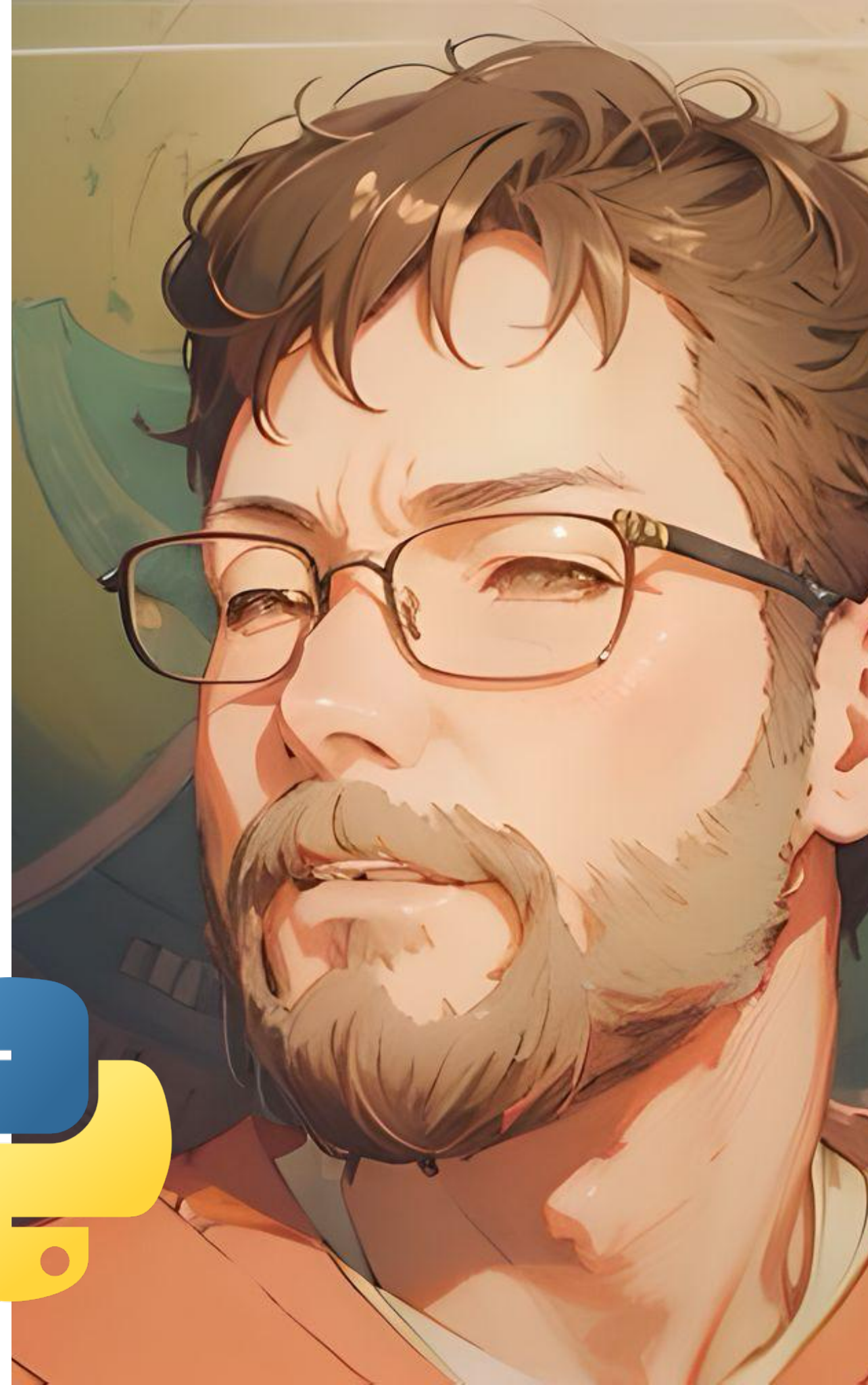
Scratch is now used in many other technologies, such as educational robot programming software (MeetEdison, mBot, LEGO Spike Prime, Arduino, etc.).



The Creation of Python

Python was created in the late 1980s by *Guido van Rossum* in the Netherlands, with its initial release in 1991. It is a high-level, general-purpose programming language. One of its most important features is the code's readability, which makes it particularly easy to learn, program, and maintain large programs written in it.

Python's capabilities can be significantly expanded through **modules** and **libraries**. We'll get to know additional features of Python in a later chapter when we will create shapes using commands.



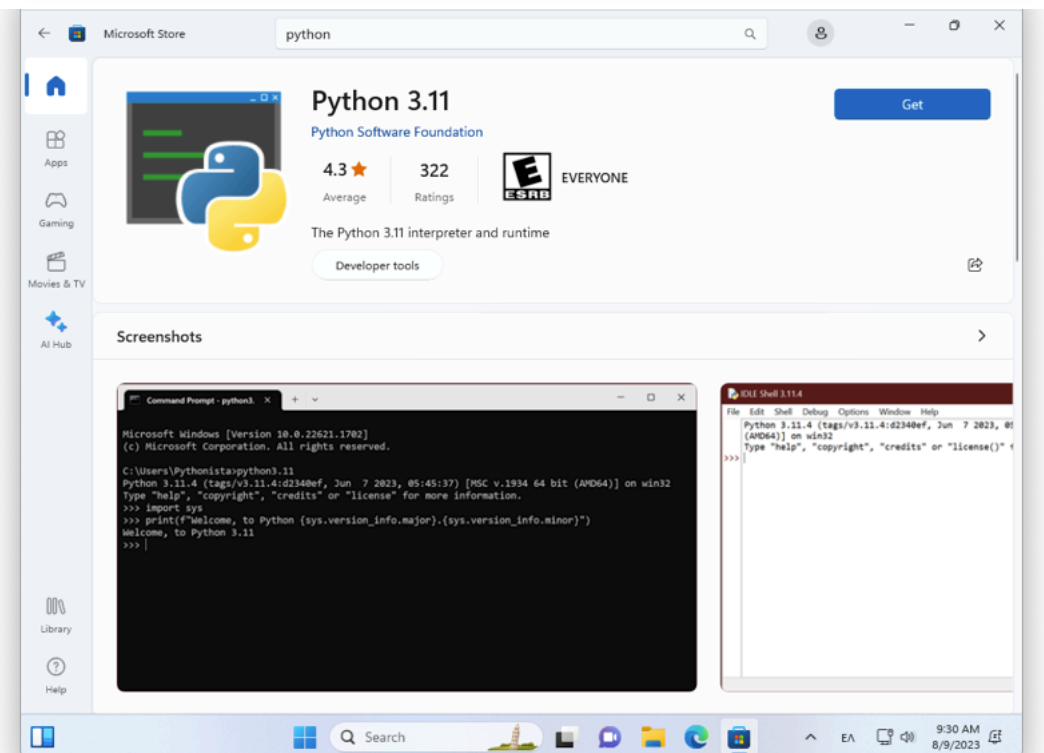
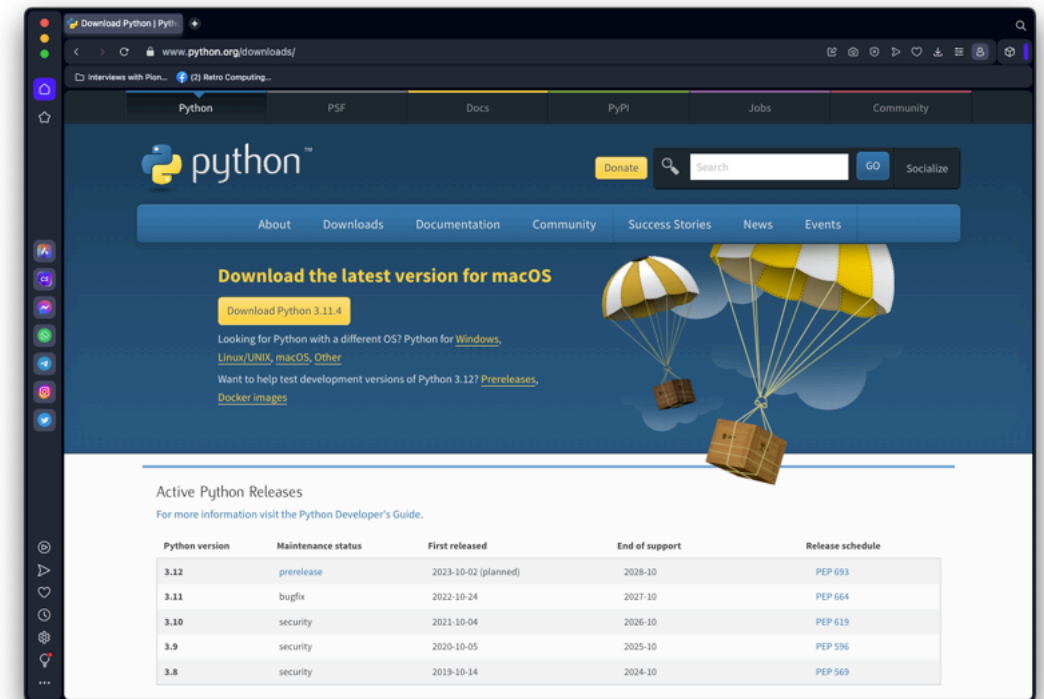
Installation of Python

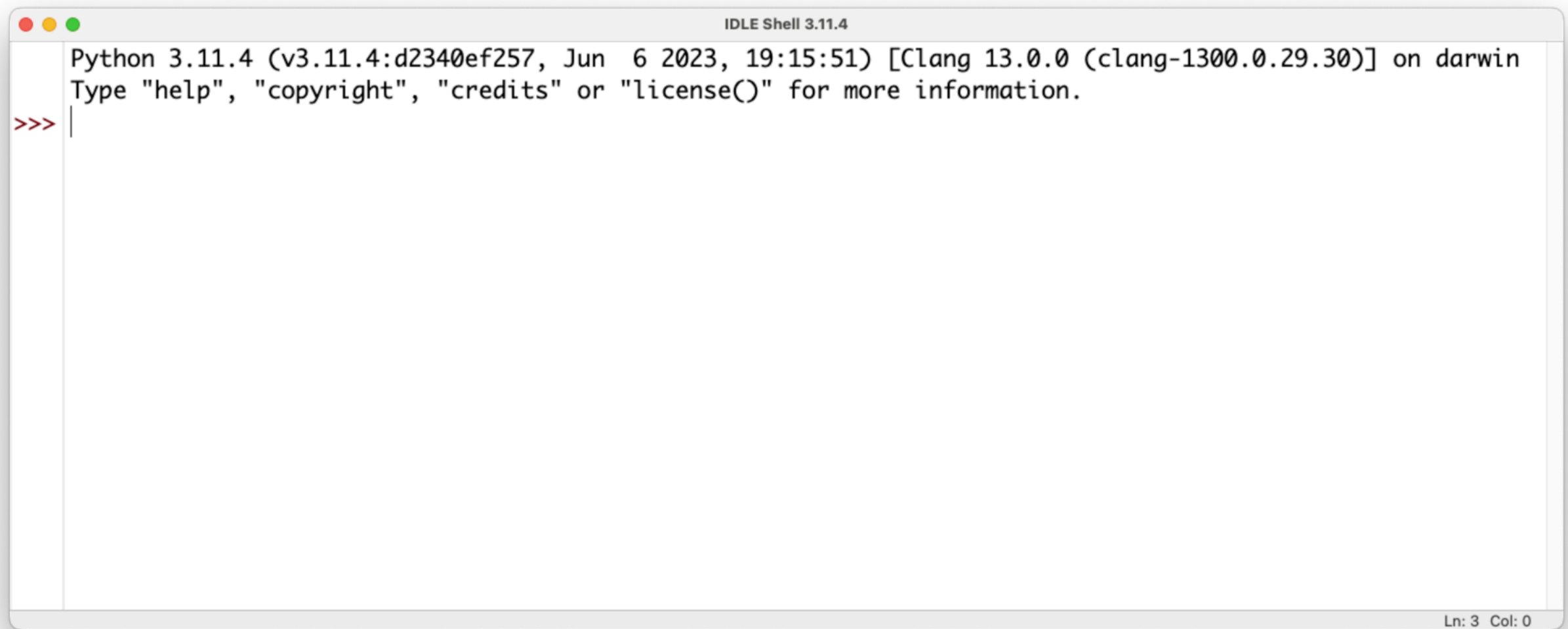
The simplest way to program in Python is by installing the necessary packages-files from the official page:

<https://www.python.org/downloads/>

From this link we can download the version for our computer (don't worry, the website will automatically detect the type of computer you are using). We should emphasize that Python and all its related packages, as well as learning and usage guides (of the website) are **completely free**.

After downloading the file, we run the installation process. On Windows, we can install Python directly from the Microsoft Store (image below).





```
Python 3.11.4 (v3.11.4:d2340ef257, Jun 6 2023, 19:15:51) [Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>> |
```

Programming Environment IDLE Shell 3.11.

The image above shows the contents of the Python programming environment. The first line shows the version of Python we are using, as well as the operating system environment.

In the last line the symbols **>>>** appear (in the margin) . After them we can type our commands in Python.

Next, we will get to know basic Python commands.

What have we learned so far?

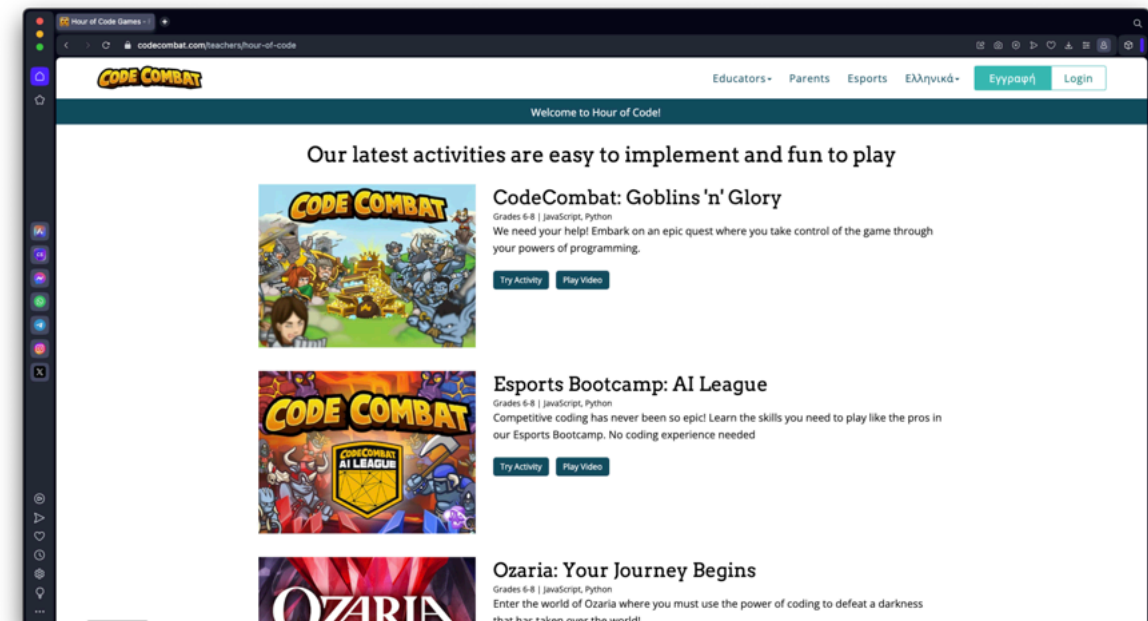
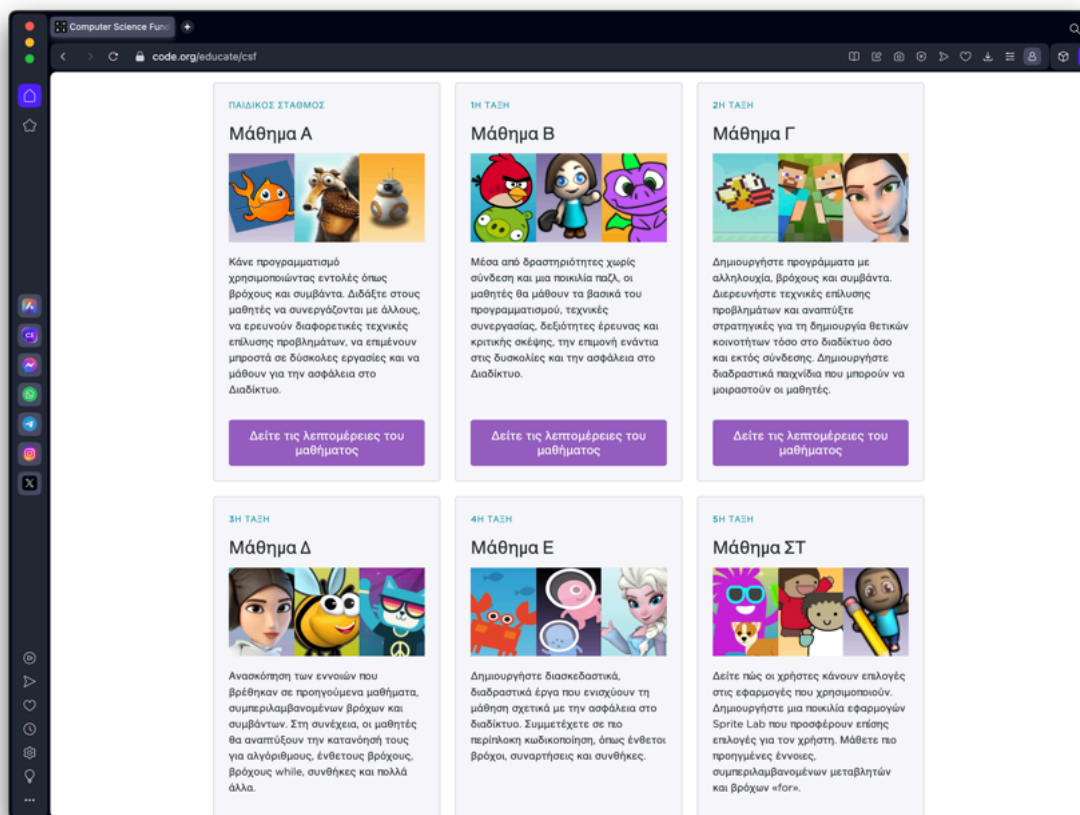
- We have learned about programming languages, which allow humans to communicate with computers.
- We read about programming languages and environments used in education.
- We have gained knowledge about the history of the Python language
- We have installed Python on our computer.



Activities

- Try **Scratch Jr** and/or **Scratch**. They can be both installed on tablet or on your computer via Google/Play Store (or Windows Store), while Scratch also “runs” online: <https://scratch.mit.edu>
- A good introduction to programming is code.org activities. Try the introductory activities from the following link: <https://code.org/educate/csf#pick-a-course>

- **All Can Code**: an online introductory programming game. You can try it from here: <https://runmarco.allcancode.com>
- **CodeCombat**: an online role-playing game based on Python (and other languages). It helps instruct the characters we control. Through a series of stages, we get to know the syntax (albeit in a simplified form) of commands in Python. The “Hour of Code” version allows us to try activities from the Ozaria game as well. <https://codecombat.com/teachers/hour-of-code>





PART B: PYTHONS & CODE

2. Let's learn Python

```
10 | print ("I don't think")  
20 | print ("Then you shouldn't talk")  
30 | #Alice in Wonderland
```


Primary commands

You must be anxious to give your first commands. Before doing this, we should explain, for the next few pages, how we will be working in an "**interactive**" mode of operation. What does this mean; Python will execute each command we type, **one at a time**. This is surely not convenient, if we want to create a program with several commands. But it's an ideal way to get started and get to know Python commands, as well as programming logic.

But be careful: programming languages recognize commands exactly as they should be written. In short, if we make spelling mistakes in a command, then our program will not execute it.





What we are going to learn:

In **Unit 2: "Introduction to Programming"** we are going to learn about:

- Python's interactive environment, where one command is executed at a time.
- Using `print()` to display information on the screen (text and numbers).
- Using Python as a calculator.
- Variables, types of variables and how we use them.
- Lists, and how we use them, as well as the differences between lists and variables.
- The "if..." condition and making decisions with it.
- Input data with the keyboard with `input()`.




```
IDLE Shell 3.11.4
Python 3.11.4 (v3.11.4:d2340ef257, Jun 6 2023, 19:15:51) [Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Hello World")
Hello World
>>>
```

Note the differences: the parenthesis, in the second case, appears in green, not black.

```
IDLE Shell 3.11.4
Python 3.11.4 (v3.11.4:d2340ef257, Jun 6 2023, 19:15:51) [Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Hello World")
Hello World
>>> print("Hello World)
...
SyntaxError: incomplete input
>>>
```

Let's look at our first command. Notice how the programming environment **colors** the commands.

The `print()` command does not print anything on paper. It shows messages on our screen. In the example it displays the message **"Hello World"**. In short, whatever we put in the parentheses, when we press the ENTER key on our keyboard, will appear on the screen.

You'll notice that **Hello World** doesn't appear in quotes. And quite rightly it is not displayed, as the quotation marks only serve to "tell" Python what text to display on the screen.

We will now try the same message, but without the closing quotes. Let's see what happens...

Because we didn't close the quotes, Python thinks we're still typing text to display on the screen. When we pressed the ENTER key on the keyboard, the command wasn't executed because there is a problem with its syntax!

Print() command so far:



With `print()` we can display text (or strings of characters) on the screen. In order for the command to be executed, our text inside the parenthesis must be in quotation marks.

```
>>> print("This is just a text")
```

As we have seen, each command is executed by pressing ENTER on the keyboard. If we want to display more lines of text, the simplest way is not to close the parenthesis. Press ENTER at the end of the line and type the next phrase/line. Every time we press ENTER (without closing the parenthesis), it simply changes the line without executing the command.

```
*IDLE Shell 3.11.4*
Python 3.11.4 (v3.11.4:d2340ef257, Jun 6 2023, 19:15:51) [Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more in
formation.
>>> print("This is the first line"
...     "This is the second line"
...     "This is the third line")
Ln: 5 Col: 31
```

In the image above, notice the **3 dots** in the box. This indicates that the command has not been executed yet. But when we close the parenthesis, the command is complete and - by pressing ENTER - will be executed.

In execution, one line will not appear below the other. It will appear one after the other:

This is the first line
This is the second line
This is the third line



Later we will see how we can display text on different lines.

Working with numbers

So far we have used the `print()` command to display text on the screen. Let's see what happens if, instead of text, we use numbers.

```
>>> print(1234)
1234
>>> print("1234")
1234
```

See the example above: in the first line, inside the parenthesis we had the number 1234 without quotes. On execution, the number of the bracket appeared.

In the second command "1234" was in quotes. It immediately appeared in **green**. When the command was executed, the number 1234 appeared again.

The result seems to be the same, but there is a big difference, which we will see next!

Let's look at the example again. This time we are going to use the addition sign.

In the first line, as shown in the image below, we have the numerical sentence in quotation marks. Python displays whatever is inside the quotes exactly as we typed it.

```
>>> print("12+5")
12+5
>>> print(12+5)
17
```

In the second command, which we haven't used quotes, Python thinks that we want to display on the screen the **result of the operation**, not the text "12+5". So it shows us the sum of 12 plus 5, which is **17**.

Priority of operations:



Python executes the operations in the correct mathematical order (priority of operations). First operations in **parentheses**, then powers, then **multiplications and divisions** and finally **additions and subtractions**.

```
>>> print((10-2)*2)
16
```

Let's see some examples of mathematical operations:

Addition: `print(100 + 5)`

Subtraction: `print(100 - 5)`

Multiplication: `print(100 * 5)`

Division: `print(100 / 5)`

Powers: `print(5**100)`

If we want to make the result displayed on the screen more **interesting**, we can **combine** the mathematical statement with the result of the operation.

```
>>> print("12+5=", 12+5)
```

When the above command is executed, **12+5=** enclosed in quotes will be displayed on the screen. Then there is the comma (,) which indicates that there is another section within the parentheses that needs to be executed. Because the following part is unquoted numbers, Python will do the operation and display the result:

```
12+5= 17
```

Python, The calculator!

Python can be used to perform mathematical operations. In the previous pages we saw how Python works, as well as the use of the `print()` command. In fact, the specific command is not needed to perform the actions. We can **directly give the actions** we want, as in the image below.

```
IDLE Shell 3.11.4
>>> 12+2+3
17
>>> 10+4*(12**3)-3*(12/4)
6913.0
>>> 12.33+2
14.33
>>> 0.22-0.011
0.209
>>> |
```

We should only pay attention to the way we write decimal numbers: in Python we always use the symbol `"."` to separate the integer part from the decimal. If we use the symbol `","` Python separates the numbers and adds them in pairs.

Let's look at an example:

```
>>> 1, 2 + 3, 8
(1, 5, 8)
```

If we look at the above operation, Python performed addition on the digits between the `"+"` symbol and separated the first digit (1) that was to the left of `","` and the last digit (8) that was to the right of `","`.



In Python we always use the `"."` symbol to write **decimal** numbers.

In operations with decimals, the addition is always done according to the position of each digit (units to units, tenths to tenths, centimeters to centimeters).



Lets get to know variables and, later, functions, which will allow us to write more complex code.

We'll also use math to perform iterations, which we'll see in later chapters.

Introduction to variables

So far, we've seen how to display text and numbers on the screen. We also learned how to use Python to perform mathematical operations. Things get a lot more interesting when we get to know the variables!

The question is "*What is the variable?*".

We should think of variables as "boxes" in computer memory, into which we place (or remove) objects. An object can be a piece of paper with **a name**. Or **a number**. Or **a phrase**. What we need to know is that, the variable can hold **one thing at a time!**

To create a variable, we simply name it and give it a "value" (content). For example:

```
>>> name="George"
```

When we press ENTER, nothing seems to happen (no message appears on the screen). In fact, we've created a variable that we call **name** (we could have called it anything, even SuperMarios). Then with the "=" sign we give a value (a content) to the variable. Because the content we're going to render is text, it's enclosed in quotes.

Then we can type the command:

```
>>> print(name)
```

In the above command, we don't need to put the name in quotes, because it is a variable. In essence, we are asking Python to display the contents of the name variable on the screen.

The word George (the content of the name variable) appears on the screen.

George

If we **didn't use quotation** marks in "George", then Python would think that inside the name variable we want to put the content of **another variable named George**. Because (currently) such a variable does not exist, an **error** message is displayed.



```
>>> name="Marios"  
>>> name="Eleni"  
>>> name="Kostas"  
>>> print(name)  
Kostas
```

As we mentioned on the previous page, in a variable we can have one piece of information at a time. Let's look at the example below:

In the first line we create the name variable and put the word Marios in it. In the second line, in the variable name, we put the word Eleni. In the third line, in the variable name, we put the word Kostas. When we execute the command `print(name)`, the word Kostas is displayed.



Every time we give a new value (a new content) to a variable, we replace its previous content with the new one.

We can work with many variables simultaneously. In the following example, we have one variable for first name and one for last name. With the command `print(name, surname)` we display the content of both variables in one line.

Note the (,) is necessary to display the contents of more than one variables in the same command.

```
>>> name="Alexis"  
>>> surname="Ioannou"  
>>> print(name, surname)  
Alexis Ioannou
```

In a similar way, we can assign a numeric value to a variable.

```
>>> arithmos=10
```

In the example above, we created a variable named `arithmos` and gave it the value 10, which is a number, so it doesn't need quotes. Now let's see how to use variables in mathematical operations:

```
>>> print(arithmos+5)
```

15

With the `print` command, we asked it to add the contents of the variable `arithmos` (which is 10) to the number 5. So the result is **15**.

In earlier times, it was common for teachers to impose punishments like "write 100 times and I won't speak in class again". Fortunately, today there are no such punishments! But if, in the time of our grandparents, there was Python, then things for students would be much simpler. They could simply do the following:

```
>>> mytext="I will not speak in class again"
```

In the variable mytext, we have placed the phrase "I will not speak in class again". Now we will see how we can repeat it with a simple command:

```
>>> print(mytext*100)
```

The command above displays the contents of the variable mytext, while the symbol * and the number 100 repeats its display on the screen.

```
>>> print(keimeno*100)
δε θα ξαναμιλήσω στο μάθημαδε θα ξαναμιλήσω στο μάθημαδε θα ξαναμιλήσω στ
ο μάθημαδε θα ξαναμιλήσω στο μάθημαδε θα ξαναμιλήσω στο μάθημαδε θα ξαναμ
ιλήσω στο μάθημαδε θα ξαναμιλήσω στο μάθημαδε θα ξαναμιλήσω στο μάθημαδε
θα ξαναμιλήσω στο μάθημαδε θα ξαναμιλήσω στο μάθημαδε θα ξαναμιλήσω στο μ
άθημαδε θα ξαναμιλήσω στο μάθημαδε θα ξαναμιλήσω στο μάθημαδε θα ξαναμιλή
σω στο μάθημαδε θα ξαναμιλήσω στο μάθημαδε θα ξαναμιλήσω στο μάθημαδε θα
ξαναμιλήσω στο μάθημαδε θα ξαναμιλήσω στο μάθημαδε θα ξαναμιλήσω στο μάθη
μαδε θα ξαναμιλήσω στο μάθημαδε θα ξαναμιλήσω στο μάθημαδε θα ξαναμιλήσω
```

Variables can also take the value of another variable.

```
>>> firstnumber=10
```

We have created a variable named firstnumber and given it the value 10.

```
>>> secondnumber=firstnumber
```

We have created a variable named secondnumber and given it the value contained in the firstnumber variable.

```
>>> print(secondnumber)
```

10

Variables can take (almost) any name we want. There are **exceptions**. For example, we cannot use "print" as a variable name. The name can also be a letter (eg N="School"). But not a number. We will see more later.



Working with variables

Pythons are especially smart! The same goes for Python. In other programming languages, we need to say in advance whether a variable will contain text, an integer, or a decimal. Python recognizes the type of the variable itself.

Let's look at an example:

```
>>> myaddress="20, Ioannou Str"
```

In the above variable (myaddress), we have text and number (the street and its number). Variables with similar content are of type String.

```
>>> myage=48
```

In the above variable (mage), we give an integer as a value. These variables are of type Integer.

```
>>> myheight=1.82
```

In the above variable (myheight), we give a decimal number as a value. These variables are of type Float (floating point).



Variables can have different names, as we saw on the previous page. We should take care that their names are meaningful. This makes it much easier to read the code to understand which variables serve which purpose.

If, for example, we want to have a variable that holds the score in a game, it would be easier to name it "score" or "myscore" (or something similar). We could simply call it S or S2 (if we have different scores).

When naming variables, we should also pay attention to **capitalization**. See the following example:

```
>>> myage=48
```

```
>>> myAge=24
```

In the example above, we seem to have the same variable (myage). But, in the second line, we wrote A in capital letters. For Python, they are two **different** variables.



Now that we've gotten to know variables better, let's do some more math with them. We already saw examples where we multiplied a variable containing text by an integer.

```
>>> programLang="Python"
>>> print(programLang*4)
```

PythonPythonPythonPython

What happens when we add an integer to a variable that contains text?

```
>>> print(programLang+5)
```

When we press ENTER to execute the above command, we get the following message:

```
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in
    <module>
      print(programLang+5)
```

```
TypeError: can only concatenate str
(not "int") to str
```

We cannot add an **integer (int)** to a variable containing **text (str)**.

But let's look at something different:

```
>>> programLang1="My favorite language is"
>>> programLang2="Python"
```

In the above commands we have created two variables. Now let's see how they appear with execution:

```
>>> print(programLang1+programLang2)
```

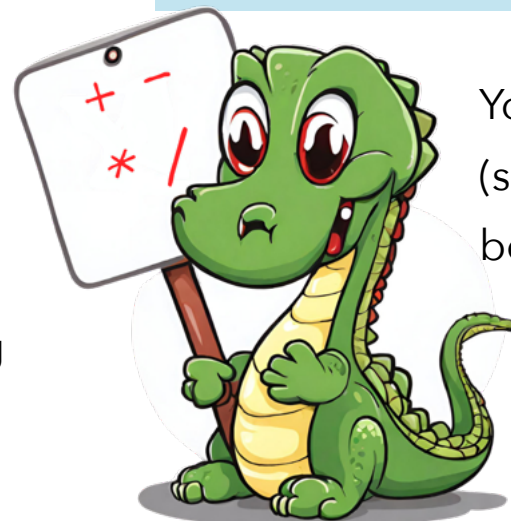
My favorite language isPython

The "+" sign has joined the contents of the two variables. However, he has not left a space between the text of one variable and the text of the other variable. In such a case, if

we want a distance, we use "," instead of "+".

```
>>> print(programLang1,programLang2)
```

My favorite language is Python



You can also practice with other symbols (subtraction, division) to see how Python behaves with variables!

Working with lists

One day Python decided to invite friends for dinner. The fridge and cupboards were empty so he had to go for shopping. Being forgetful, he, somehow, had to remember many different things to buy. So he made a list.

Variables, as we saw in previous pages, can hold one piece of information at a time. For example, the variable

```
myfood="banana"
```

holds the value (content) "banana" and nothing else.

Python could, of course, use a variable that contains in quotes all the things he wanted, e.g:

```
myfood="banana, strawberry, raspberry".
```

The problem is that he will get confused: he will be looking to find ONE box in the supermarket that has all this in it.

So, variables are good, but if we want to store a lot

of (and different) information, so that we can find each one separately, we will use **lists**.

A list has the following form:

```
>>> mylist=["banana","strawberry","ice cream"]
```

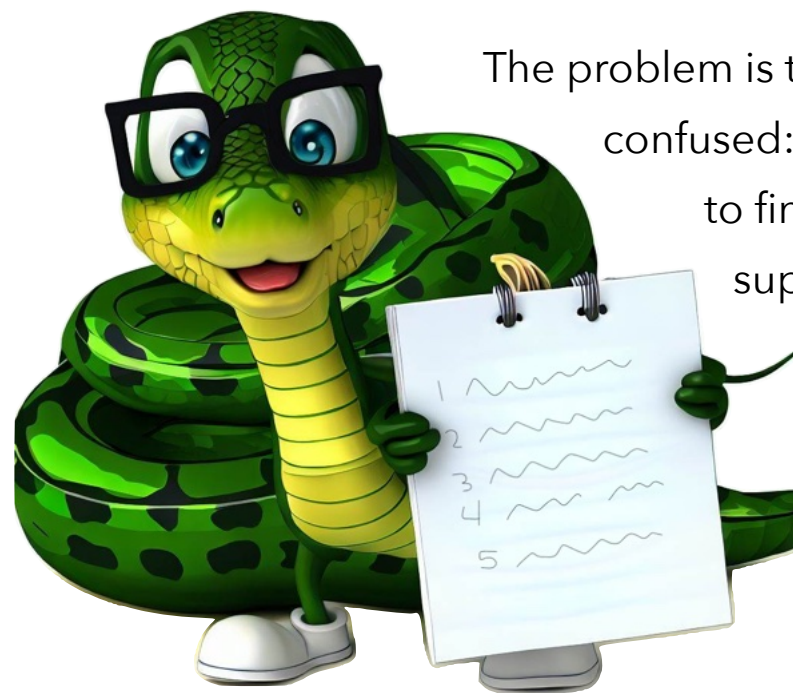
We give a name to the list, as we would give it to the variable ("mylist" list) above. Then we type "=" to give the contents of the list. If it will be text (string), we write each object separately with quotation marks. A list, unlike a variable, has its contents enclosed in "[]" symbols.

To see the contents of the list, type the command:

```
>>> print(my list)
```

```
['banana', 'cherry', 'strawberry']
```

Each item in a list has its own **order**: the first item is at position **0**, the second at position **1**, the third at position **2**. In the next chapter we will see the usefulness of the list and the positions of the objects.



"Lists", 'Lists', Lists!

Let's examine the lists below:

```
mylist=['banana','icecream','strawberry']
```

```
mylist=["banana","icecream","strawberry"]
```

In both cases, the result seems to be the same: mylist contains the same information as string.

In case we want to use actual numbers (integers) in the list, it should have the following format:

```
mynewlist=[10,14,23,44]
```

To show the contents of a list, you don't necessarily need the print() command. **Print()** is useful when we want to display **other information** along with the contents of the list.



We can simply type the name of the list to display its contents:

```
>>> mynewlist  
[10,14,23,44]
```

If we have two lists, we can join their contents. For example:

```
>>> mylist1=["banana","watermelon"]
```

```
>>> mylist2=["apple","cherry"]
```

```
>>> mylist2=mylist2+mylist1
```

In the last command, we put the contents of mylist2 together with the contents of mylist1. If we type mylist2, we get the following:

```
>>> mylist2  
['apple', 'cherry', 'banana', 'watermelon']
```

We can add content to a list with the "+" sign and the items inside []. For example:

```
>>> mylist2=mylist2+["mango"]
```

The word **mango** will be **added** to the **end** of our list.



Decisions...

Pythons are intelligent animals. A python wanted to upgrade his computer. He wanted to see if his memory was more than 8GB. He thought "IF (the computer's) memory is less than 8GB, I'll upgrade it". Before making the decision to upgrade his computer's memory, he should first check if it was less than 8GB (and, you know, with less than that, you don't run fast Windows 11 or MacOS X or new versions of Linux).

How would we test this (almost) with pseudocode:

"If the memory (of the computer) < 8GB upgrade memory"

So, first we will check if our memory is smaller (using the "<" symbol) than 8GB. **If** this is the case, **then** and only then will we **upgrade** the computer's memory.

At this point we are going to use a variable (mymemory) to which we'll give the value 8. Let's see how our computer will check it.



Lets write the next code:

```
>>> mymemory=8
```

We have created a variable to which we give the value 8 (our computer's memory in GB).

```
>>> if mymemory<8: print("You need to upgrade  
your memory")
```

The new "if" command asks us to check if a condition is true. In this case, we ask it to check IF the value of the variable is less than 8. If it is less than 8, then the message Upgrade memory will be displayed.

In the example above, we only used one condition (memory<8). So we only used one line for the condition code.

Usually in conditions we follow the following structure:

```
memory=8  
if mymemory<8:  
    print("You need to upgrade your  
memory")
```

The result is of course the same, since we again have only one condition to check. Then we will see other examples.

Until now, we checked a number and returned only one response if the condition was true. But if it is not the case, why not give some information? Here we will use the "else" command.

Let's issue the following command:

```
print("Your memory needs upgrade") if  
memory<8 else print("No need to upgrade")
```

In the example above, we followed a different syntax. In one line we request that the message appear on the screen **Your memory needs upgrade** IF the value of mymemory is less than 8, **ELSE** display the message **No need to upgrade**.



We will study the conditions in the next chapter, in greater depth.

It's important to understand that our examples so far have involved executing one command at a time. Later we will learn how to create a complex program consisting of many lines of code.

In the previous pages, we got to know the "if" command and studied some simple examples. It is important to see, next, how we can use this command for more complex comparisons.

We will initially start with two variables:

```
number1=10  
number2=20
```

We assigned different values to these two variables.

We want to compare the contents of the two variables to see if they are the same.

```
if number1 == number2: print ("Equal")
```

With the above command, we check if their content is the same. This is achieved with the double "=" sign. In the above example, the condition does not apply, so no message will be displayed.

If what we are interested in is checking if their value is different, then we use the command:

```
if number1 != number2: print ("Equal")
```

Checks we can do:

- One variable is less than another:
number1 < number2
- One variable is greater than another:
number1 > number2
- One variable is less than or equal to another:
number1 <= number2
- One variable is greater than or equal to another:
number1 >= number2

For comparisons, we can also use **And**, **Or** and **Not**.

```
if number1=100 and number2=100:  
print("Equal")
```

In the command above we check if both conditions apply (both one number and the other are equal).

```
if number1=100 or number2=100: print("One hundred")
```

In the command above, we check if even one of the two is equal to 100.

What have we learned so far?

- In this chapter we have learned general information about programming languages and focused on Python.
- With the `print()` command we can display text and result of operations and variables on the screen. The text must be in quotation marks `" "`.
- Mathematical operations (e.g. `5+7`) are performed automatically by pressing ENTER without using `print()`.
- Variables can be used to store characters (strings), integers (integers) and decimals (floats).
- We used lists and learned their differences in relation to variables.
- With `if` command we can check whether a condition is true or not.



Activities

1. What is the result of running the code below?

```
number1=10
number2=5
name=input("What's your name?")
print("Welcome",name)
number=5
print("The addition of the two numbers
is", number1+number)
```

2. In the code below

(a) tick the information that is not necessary,

(b) list the result of its execution:

```
numb1=10
numb2=4
numb3=2
numb4=8
#Operations using the above variables
```

```
print("The result
is",numb1+numb3+numb3*2)
```

3. The program below must add the values of the three variables and display their result.

(a) Fill in the missing code.

(b) Write the result of running the code.

```
print("Addition of the integers")
a=20
b=15
c=30
print
```

4. Correct the code below. Then write the result of its execution.

```
number1=5
number2=2
print("The addition is, number1+number2)
print(The multiplication is number1*number)
```

3. “Many many commands”

```
10 | print ("Who in the world am I?")  
20 | print ("Ah, that's the great puzzle!")  
30 | #Alice in Wonderland
```


Program...

Pythons are quite intelligent animals, as we have mentioned in the previous chapters! So far, we've worked with one command at a time. Pythons, however, can work with multiple commands. In this way we can create complex programs, which, however, perform many different tasks.

In this chapter we will learn new commands, which we will combine to create our first program.





What we are going to learn:

- In **Unit 3 "Many many commands"** we are going to work with multi-line code.
- We will use `print()` to display multiple lines of information on the screen at once.
- With the command `input()` we will give a value to a variable from the keyboard.
- With `if...else` we will check if a condition is true (e.g. `10 > 5`) so that our program can "decide" the course of a simple game.
- With `type()` we will display the type of a variable.
- With the help of `int(input())` we will enter data from the keyboard as an integer.




```
IDLE Shell 3.11.4
Python 3.11.4 (v3.11.4:d2340ef257, Jun  6 2023, 19:15:51) [Clang 13.0.0 (c
lang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("5 in the second power equals ",5**2)
5 in the second power equals 25
>>> |
```



Until now, we wrote and executed one command at a time. This is useful, but it doesn't help us when we want to create a complex application.

In the above example, we calculated the square of 5 (or 5 to the second power).

To create a program with several lines of code, we use code editing software (Editors). IDLE includes its own Editor. From IDLE's File menu, select New File (top right image).

We immediately notice the differences between the new window in which we will write a series of commands (top right) and the IDLE window we worked with in the previous chapters.

On the next page we will create our program and learn about new commands.



To be able to write a program (a series of commands) in Python, we can even use simple text editors! Notepad on Windows or SimpleText on MacOS are enough to write the code.

Python Files

Before we start adding commands, it's a good idea to save our file. From the File menu, select Save Us...



We name our program "myfirstpython.py" and save it. We have already taken a huge step in Python.

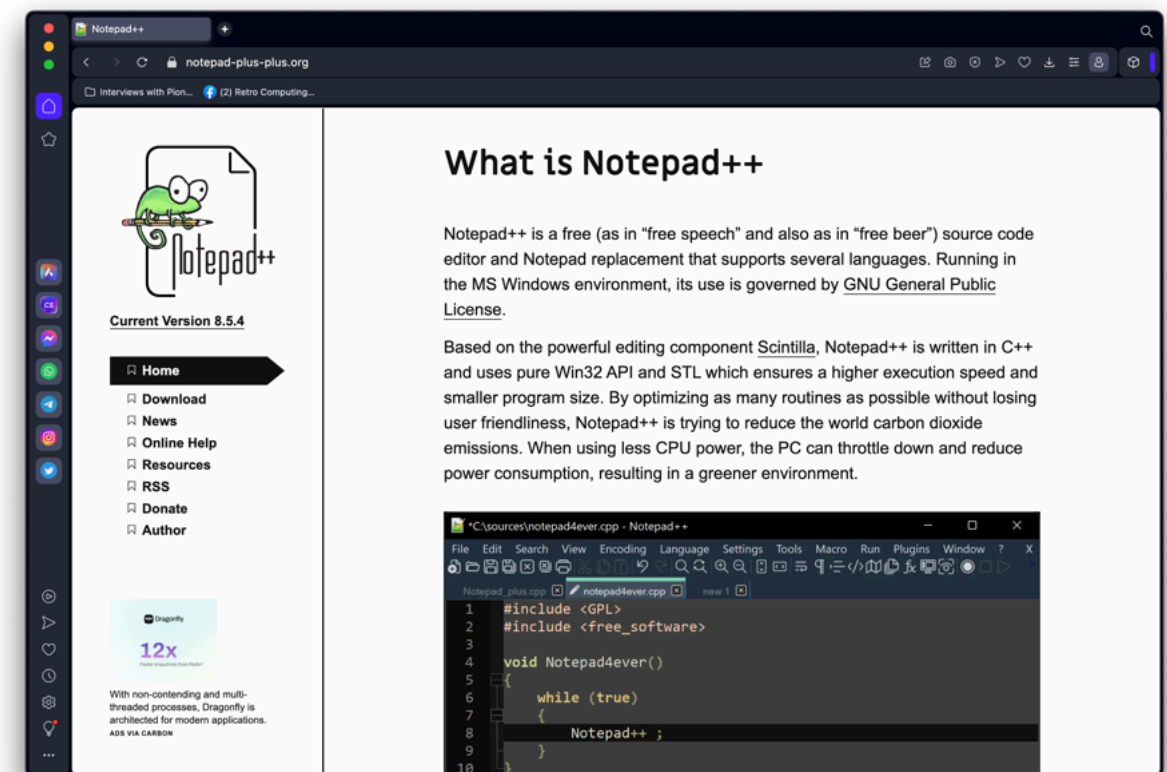


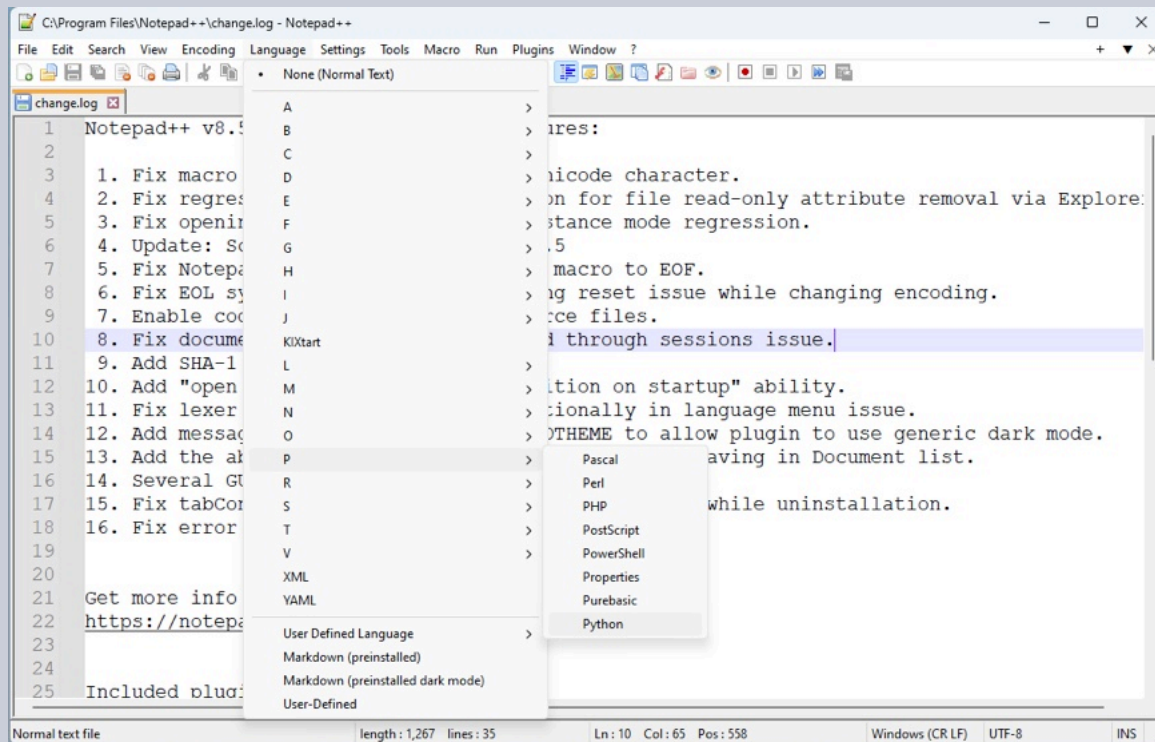
Python files have a **.py** extension, Our first file is named myfirstpython.py. It's a good idea to give our files names that tell us exactly what the program does.

Alternatively...

In addition to Python IDLE, we can use other applications, which give other features (for example, support for many programming languages).

There are a lot of software that can serve us. On MacOS, a good choice is BBEDIT. On Windows, a good choice is Notepad++ (image below). Both are free and we can download them from our computer's App Store or from their websites.



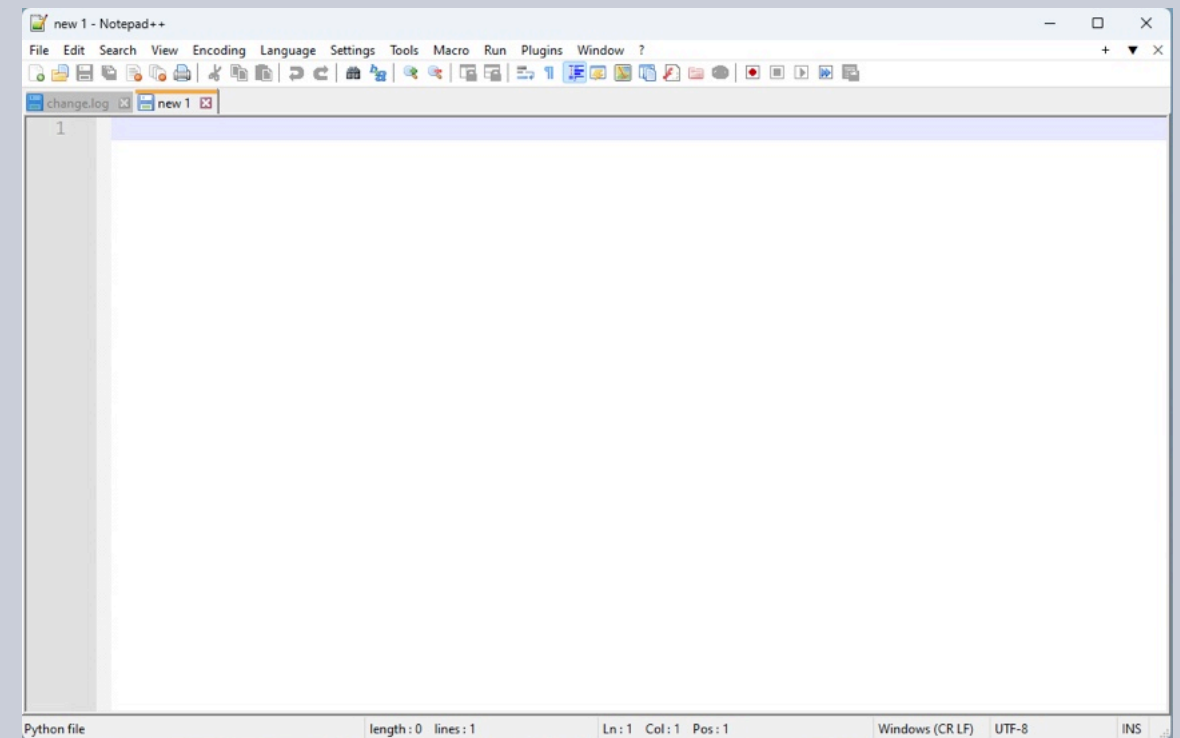


About Notepad++

If you want to try something different, you can write code in **Notepad++**, a great free Windows application.

Upon starting Notepad++, general information about the downloaded program is displayed. We are interested in "showing" Notepad++ that we want to program in Python.

When we set Python as the primary language, then the corresponding coloring appears in the commands.



From the Language menu, select Python (from the letter "P"), as shown in the image above.

To create a new file, from the File menu, select New (picture on the top right).

Notepad++ is (also) used by schoolchildren and students, as it is **free** and offers great **flexibility** with support for many programming languages.



Let's Run!



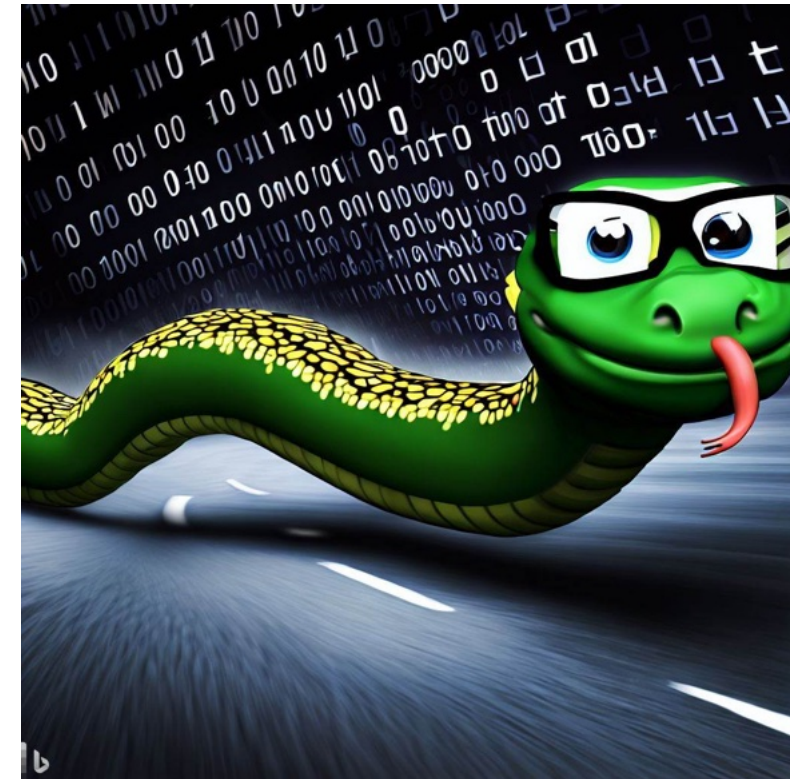
```
myfirstpython.py - /Users/akoftero/Documents/myfirstpython.py (3.11.4)
print("Welcome to the program 'Find my age'")
Ln: 2 Col: 0
```

We have written our first command. Unlike previous examples, with line break, it is not executed. From the Run menu we should select "Run Module".



```
IDLE Shell 3.11.4
Python 3.11.4 (v3.11.4:d2340ef257, Jun 6 2023, 19:15:51) [Clang 13.0.0 (c
lang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>> print ("5 in the second power equals ",5**2)
5 in the second power equals 25
>>>
===== RESTART: /Users/akoftero/Documents/myfirstpython.py =====
Welcome to the program 'Find my age'
>>> |
Ln: 8 Col: 0
```

Our program "runs" and its result is displayed (the message 'Welcome to the program 'Find my age'').



With the 'Run' command, our Python program (which we understand) is "translated" into the language that pythons (ok... computers) understand. This happens every time we select 'Run'



In the following pages we will enrich our program with new commands.


```
myfirstpython.py - /Users/akoftero/Documents/myfirstpython.py (3.11.4)
print("*****")
print("Welcome to the program 'Find my age'")
print("Guess the age of our friendly python")
print("*****")
Ln: 4 Col: 45
```

```
IDLE Shell 3.11.4
Python 3.11.4 (v3.11.4:d2340ef257, Jun 6 2023, 19:15:51) [C
lang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.
>>>
===== RESTART: /Users/akoftero/Documents/myfirstpyth
on.py =====
*****
Welcome to the program 'Find my age'
Guess the age of our friendly python
*****
>>>
Ln: 9 Col: 0
```

We have added 4 print command lines. In the first and fourth lines, we use the "*" symbol to show a box above and below our text (image above right).

Line 2 and 3 of our code (image above) gives us information about the program. In line 2 it gives us the name of the game, while in line 3 it explains what our purpose is.

Next, we'll add some more code.

```
print()
print()
print("How many years does a python live?")
```

The two print() lines we have added leave two blank lines between the introductory part of the program and the query we asked.

Next, we'll show how to give input with the keyboard.



The "*" symbol can be used in conjunction with the print() command to create shapes on the screen. We will create shapes in a later chapter of this book.



Input data

So far, we've managed to write 7 lines of code. But our program does not allow us to give the age a python might be. To give input from the keyboard, we should use a new command, input.

```
age=input()
```

Let's look at the above command: "age" is a variable. In this variable we will store the age (number) that we will give from the keyboard.

"input" is the command with which Python expects us to provide some information with the keyboard. As soon as we press the ENTER key, the information we typed (number, text) is stored in the "age" variable.

```
print("How many years does a python  
live?")  
age=input()
```



Η μεταβλητή πιο πάνω δεν έχει τη σωστή μορφή! Αυτό θα το εξηγήσουμε αργότερα, όταν θα μιλήσουμε για αλφαριθμητικές τιμές και για ακέραιους!

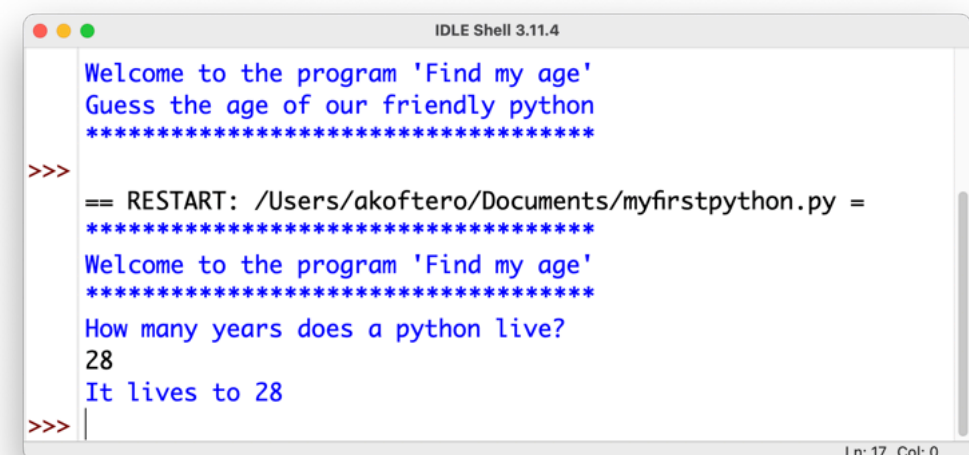
When we run the above program, the print message appears ("How many years does a python live?") and the computer expects us to type some information. Press ENTER to complete the process.

To make our program more interesting, we add one more command:

```
print("It lives to",age)
```

The above command displays on the screen the message "It lives to " followed by the number we entered with the keyboard (and stored in the "age" variable).

Then we will improve the code even more, using other commands.



```
IDLE Shell 3.11.4  
  
Welcome to the program 'Find my age'  
Guess the age of our friendly python  
*****  
>>> == RESTART: /Users/akoftero/Documents/myfirstpython.py =  
*****  
Welcome to the program 'Find my age'  
*****  
How many years does a python live?  
28  
It lives to 28  
>>>
```

Ln: 17 Col: 0

Let's look at the input() command again:

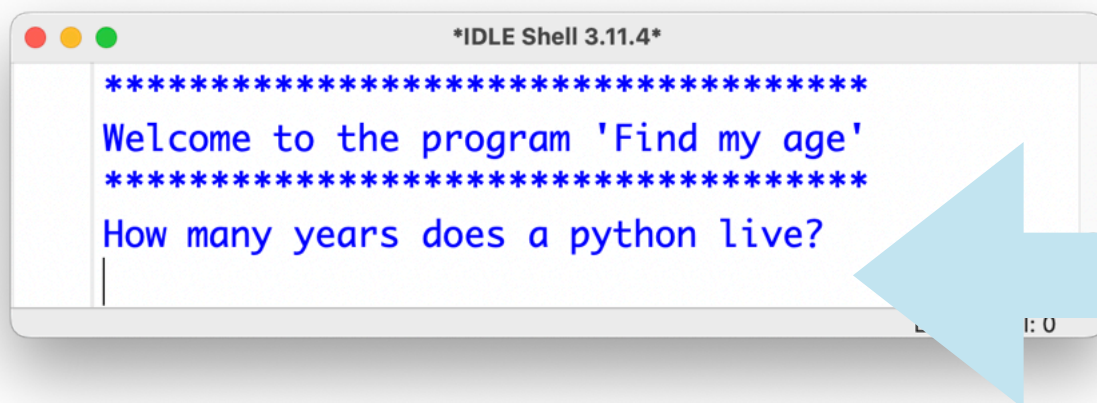
```
print("How many years does a python  
live?")  
age=input()
```

In the above example we used two commands. print() gives us the message that explains what to type. In the second command, with input() we ask someone to type a number. The number is stored in the age variable.

We could replace the two above with one command, which does exactly the same thing:

```
age=input("How many years does a python live?")
```

Within the brackets of input() we have typed the same text we previously had added to the print() command.

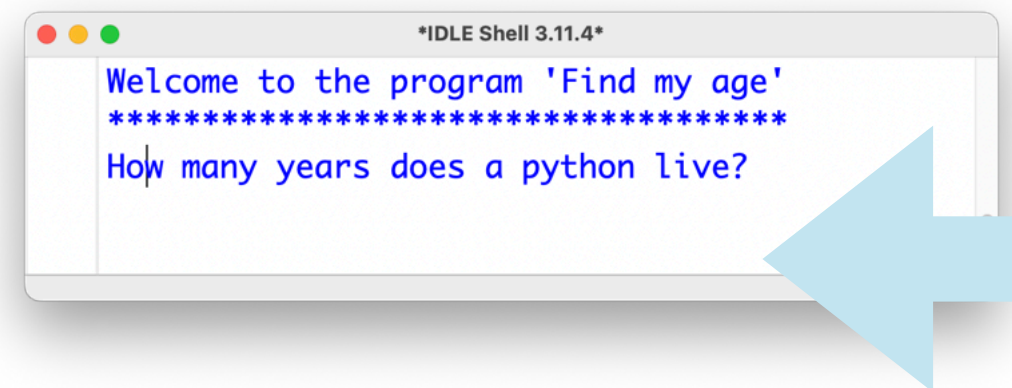


In the example above, the number entry point is to the right of the text (image lower left).

We can modify the input() command so that we type the number in the next line:

```
age=input("How many years does a python live?\n")
```

The addition of \n transfers the information we will give from the keyboard to the **next line** (image below).



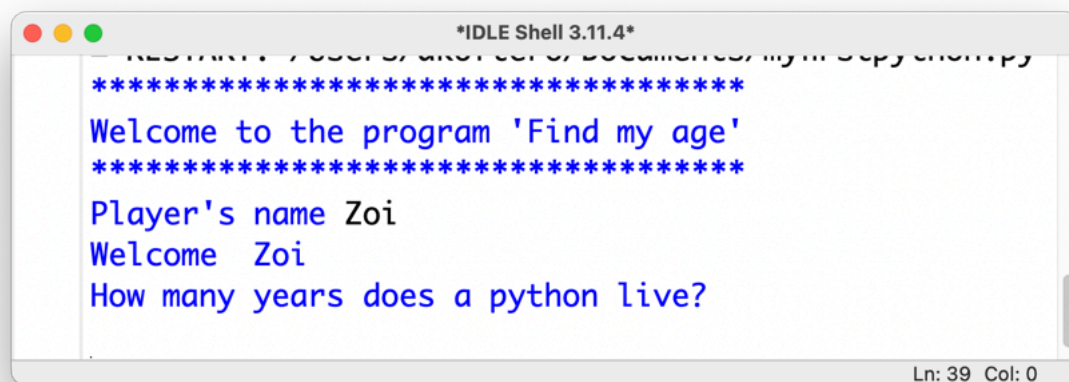
On the next page we will see other uses of the input() command.

Our beloved python can type text in addition to numbers.

```
name=input("Player's name")  
print("Welcome",name)
```

In the first command, we ask the player to write his or her name. The name will be stored in the name variable.

With the print command, the message "Welcome" is printed and then the content of the name variable (image below).



```
*IDLE Shell 3.11.4*  
*****  
Welcome to the program 'Find my age'  
*****  
Player's name Zoi  
Welcome Zoi  
How many years does a python live?
```

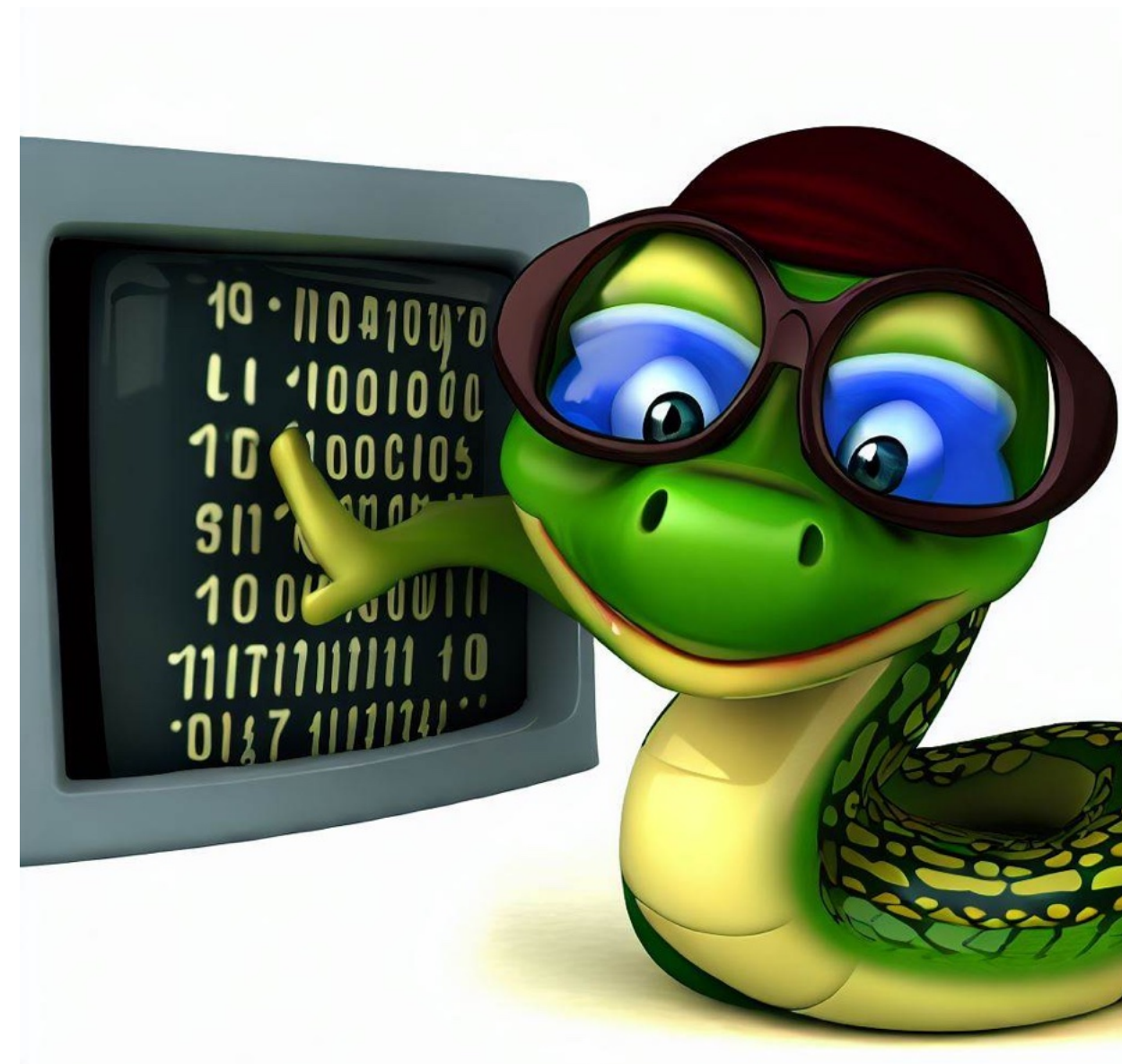
Ln: 39 Col: 0



A bit of **grammar**: when we type our name, it's in the nominative case. When the computer displays it, it continues to remain in the nominative, even though it should be in the vocative.

The input() command can be used to enter numbers with which to perform mathematical operations.

We can also use input() to select (multiple selection). And we will see this later in our book!

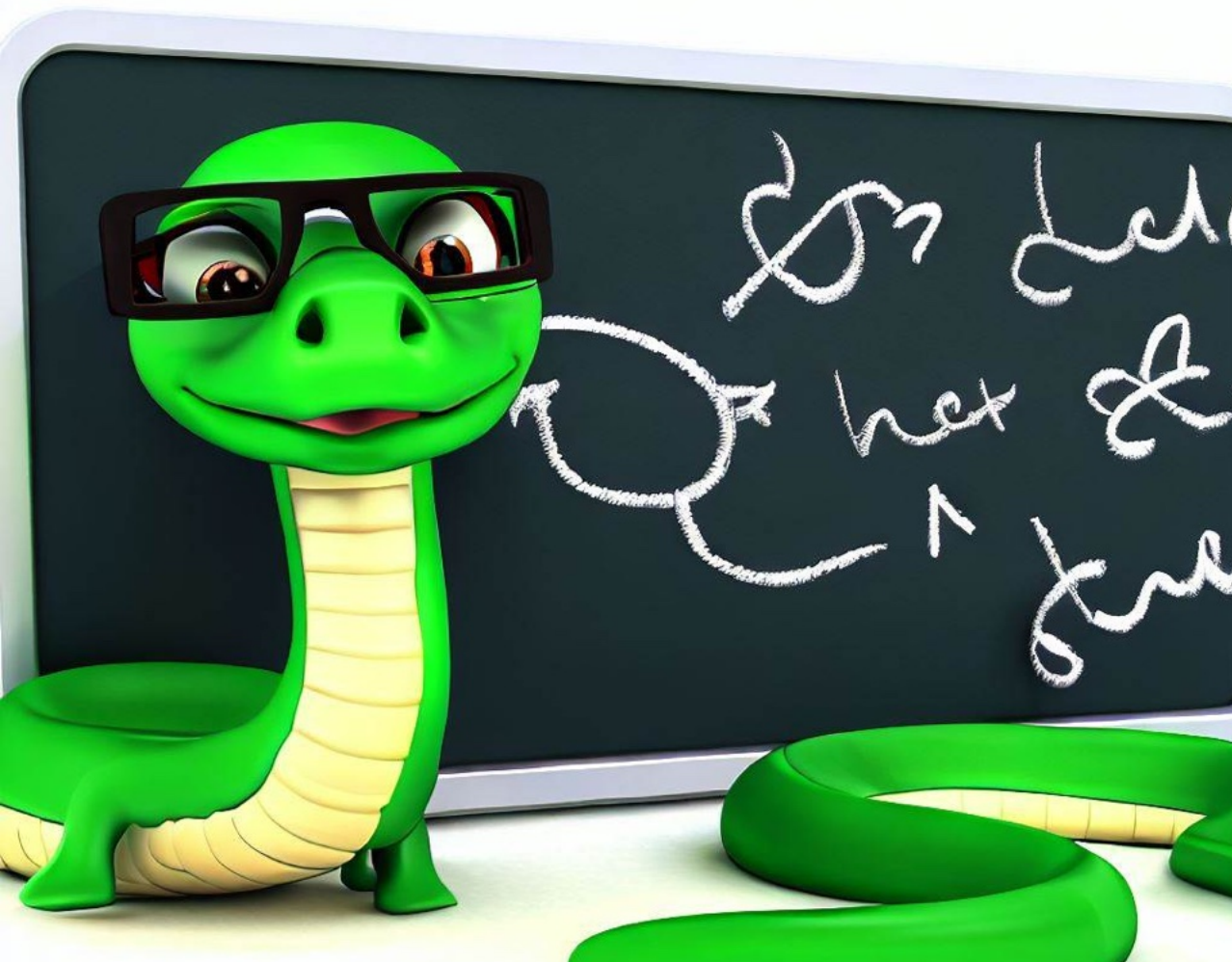


Let's comment...

Pythons are meticulous and "commentative" animals. They like to put notes in their texts. This helps them understand what each paragraph or section of text says. The same when writing computer programs.

Comments help to better understand what a program does and/or parts of a program.

To write a comment, we have to -somehow- tell Python that it is not a command to be executed, and simply ignore it.



```
myfirstpython.py - /Users/akoftero/Documents/myfirstpython.py (3.11.4)

print("*****")
print("Welcome to the program 'Find my age'")
print("*****")

name=input("Player's name")
print("Welcome",name)
#print("How many years does a python live?\n")
age=input()
print("It lives to",age)
```

Ln: 7 Col: 1

In the example above, the print() command is shown in red. We have put the "#" symbol in front of it. This symbol, at the beginning of the command, tells Python to ignore it in execution because it is a **comment**.

In the image below we see the comments at the beginning of a line (they explain what the code does) but also to the right of a command (they explain what the command does).

```
myfirstpython.py - /Users/akoftero/Documents/myfirstpython.py (3.11.4)

print("*****")
print("Welcome to the program 'Find my age'")
print("*****")
print()
print()
#With the following commands, we ask for a number
#to guess the age a python can reach
name=input("Player's name")
print("Welcome",name)
print("How many years does a python live?\n") #we ask for an integer
age=input()
print("It lives to",age)
```

Ln: 10 Col: 68

The decisive python!

Time for the friendly green python to decide if we got his age right! We should - you guessed it - use the "if" command. A python lives about 10 years.

```
if age==10:  
    print("You have found the correct  
age!")
```

On the first line, we compare the value of the variable age (given by the keyboard) to the number 10 (the number of years a python lives).

In comparison, we have the double symbol == which indicates "equal to". We also use "greater than or equal to" (>=), "less than or equal to" (<=) and "less than/greater than".

IF the condition is true (the number we gave is 10), THEN the message "You found the correct age" is displayed.



The print() command appears below and to the right of the if. All statements executed IF the condition is true must be below the if and indented.

However, we want it to display a message when we enter a different number. We should also add 'else'.

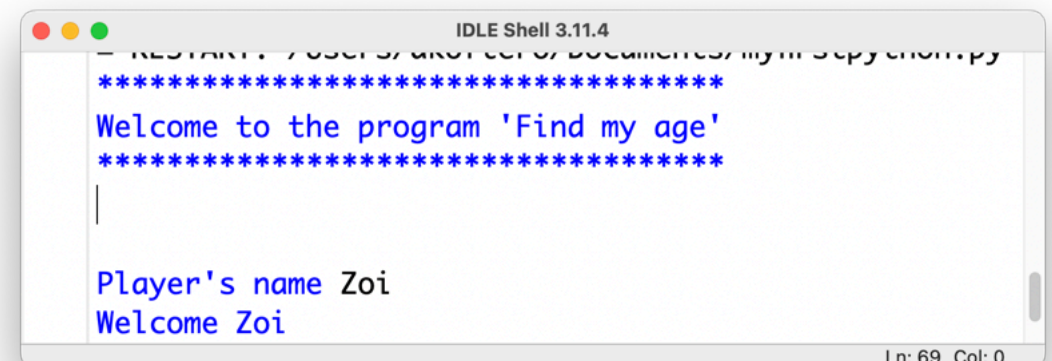
```
if age==10:  
    print("You have found the correct  
age!")  
else:  
    print("Unfortunately you were  
wrong!")
```

IF the value of the variable age is 10, THEN it will display the first message, ELSE it will display the second message.

We can write these commands in one line:

```
print("Correct!") if age==10 else print("Wrong answer!")
```

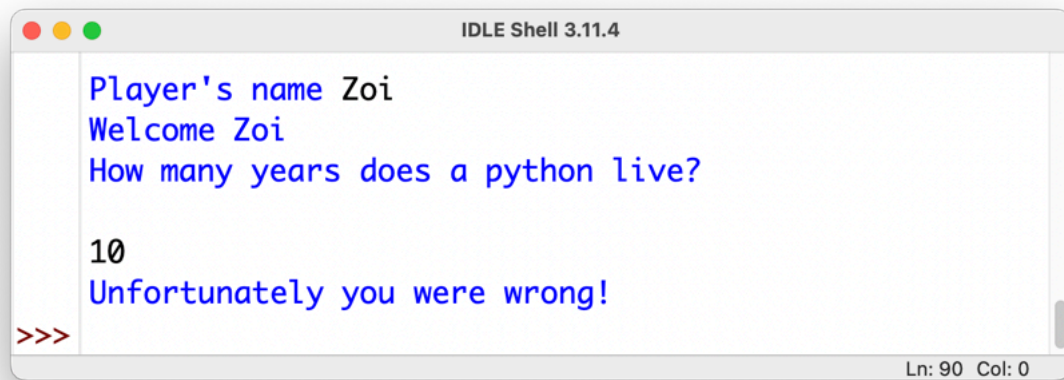
In the image below, we see the result of the program, when we give the wrong answer. On the next page we will learn how we can continue the game and give an answer again.



```
IDLE Shell 3.11.4  
*****  
Welcome to the program 'Find my age'  
*****  
  
Player's name Zoi  
Welcome Zoi  
Ln: 69 Col: 0
```


Code error!

On the previous page, we saw the message that our program outputs if we give the wrong number for the age of the python. But what happens when we give the right number?



```
Player's name Zoi
Welcome Zoi
How many years does a python live?
10
Unfortunately you were wrong!
```

As we can see in the image above, even though we type the correct answer (which is 10), it considers it wrong! But why; since our code is correct. Or is something wrong...?

Integers and oranges...

On a previous page, we talked about types of variables. Some variables, the values they take are whole numbers (integers or int). Others are text (which can also contain or

consist of numbers) and are called strings. Others accept decimals (floating point or float).

If we want to know what type a variable is, we use `type()`.

In our code we have the variable `age`, to which we give a value from the keyboard.

```
age=input("How many years does a python live?\n")
```

We then compare this variable to see if it equals the number 10.

```
if age==10:
```

But there's something we didn't notice: when we give information to input, it automatically puts it in the variable not as an integer (int) but as a text (str)!

So, whatever number we give to the input from the keyboard, it will consider it to be text and not a number and therefore the comparison will always be wrong (as if we tell it to compare whether oranges are the number 10)! Then we'll see how to fix it.

Variable type

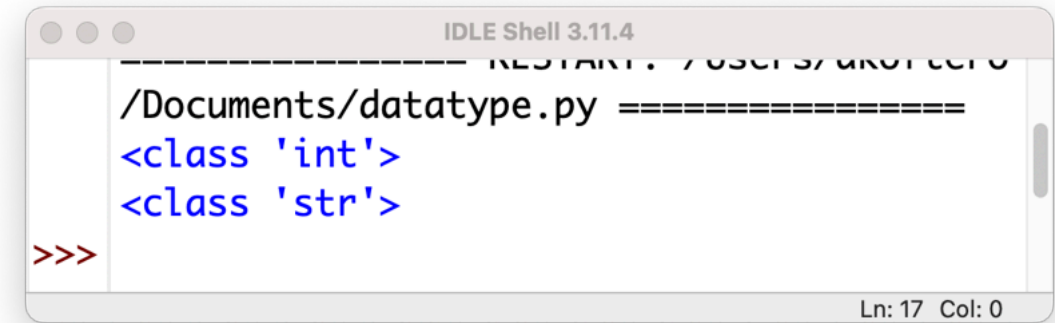
We want to give a number from the keyboard so it can compare it to the python's age and see if we got the right answer or not.

The question is: how do we tell Python that the number we want to return is an integer and not text? First of all, let's see how we detect the type of a variable...

We will use a new command, `type()` to check the type of a variable. To display the type of the variable on the screen, we'll type it inside `print()`. Let's look at the example below:

```
x=10      #in variable x we give 10 as a number
y="10"    #in variable x we give 10 as a text
print(type(x))
print(type(y))
```

`type(x)` finds the type of variable `x`. But to display its contents on the screen, we need to put it inside `print()`. The same for the variable `y`. We run the program to see the result:



```
----- RESTART: /Users/.../Documents/datatype.py -----
<class 'int'>
<class 'str'>
>>>
```

The first variable (`x=10`) is recognized as an integer (`<class 'int'>`). The second variable, although it contains the number 10, because it is in quotation marks (`y="10"`), is recognized as text (algorithm - `str`), which is why the message `<class 'str'>` appears on the second line (image above) `'>`.

Variables of type **str** (string) are used to enter **text**, but can also include **numbers** (or only numbers), but also **symbols**. That is why it is called "string".



Next we will solve the problem of entering a number as an integer from the keyboard.

Variable as Integer

If we try to give a number from the keyboard, with `input()`, it will be recognized as text (alphanumeric) and not as an integer (note: the same applies to decimals).

Let's see the command with which we give information with the keyboard, in the age variable.

```
age=input("Please give a number.")
```

We will make a change to the above code so that the value of the age variable is an integer:

```
age=int(input("Please give a number."))
```

The above change to the `input()` command solves the problem, as we are telling Python that we will be inputting a number (integer) from the keyboard.



Add the **double brackets** at the end of the command above. This is required since `input()` will be placed within `int()`.

```
int( input("Enter number") )
```

The correct code with the change is:

```
myfirstpython.py - /Users/akoftero/Documents/myfirstpython.py (3.11.4)
print("*****")
print("Welcome to the program 'Find my age'")
print("*****")
print()
print()
#With the following commands, we ask for a number
#to guess the age a python can reach
count=0 #counts number of attempts
age=1 #initial value of age
while count<3:
    age=int(input("How many years does a python live?"))#we ask for an integer
    print(age)
    if age==10:
        print("You have found the correct age!")
        break
    else:
        print("Unfortunately you were wrong!")
    count=count+1 #increases the count by 1
Ln: 17 Col: 8
```

If we now try to run the program, we will see how it recognizes 10 as the correct answer.

```
IDLE Shell 3.11.4
Welcome to the program 'Find my age'
*****

How many years does a python live?10
10
You have found the correct age!
>>>
Ln: 100 Col: 0
```


"Bugs" in the code!

Pythons, when translating from one language to another, prefer to do it sentence by sentence. Others prefer to take the text (or what a speaker has to say) and translate it in its entirety.

This has both advantages and disadvantages (which we will not go into in this book). But as Python "runs" a program, it executes the commands in sequence.



```
print("*****Hello Generator *****")
name=input("Please enter name: ")
print("Hello ",name)
print(How are you today)
print("*****Goodbye*****")
```

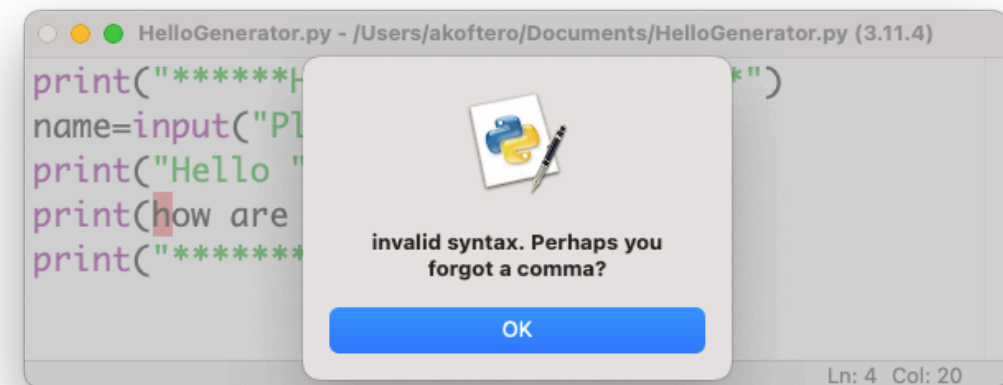
Ln: 1 Col: 0



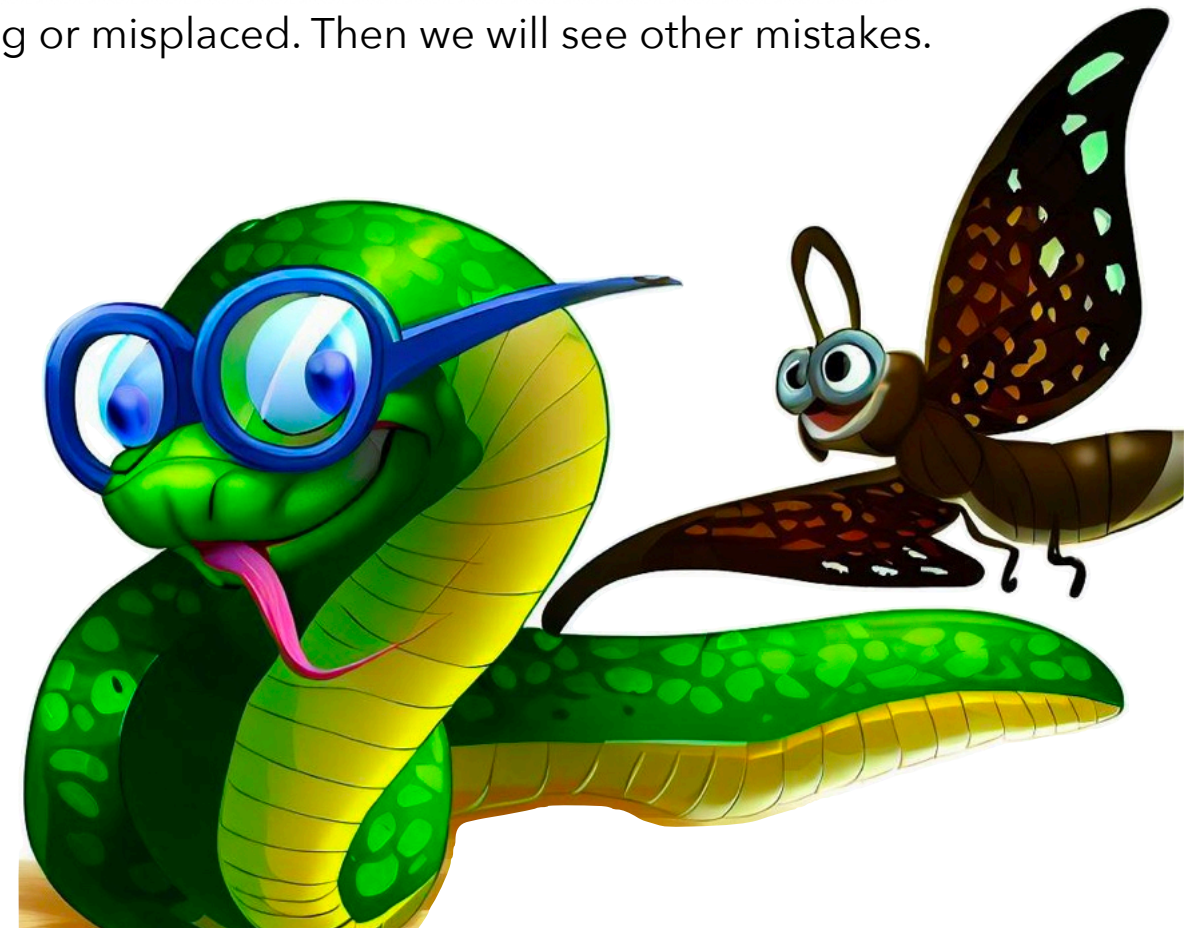
The term "**bug**" has been used since 1843 and is said to have been coined by Thomas Edison, the famous inventor. In 1947, it was used by Gray Hopper, due to a moth causing a short circuit in the Mark 2 computer.

In the image (left) we see that there is an error in the fourth line: the text is not enclosed in quotation marks.

When running the program, Python will immediately inform us of the particular error and indicate the line it is on.



The error above is syntactic, because some characters are missing or misplaced. Then we will see other mistakes.

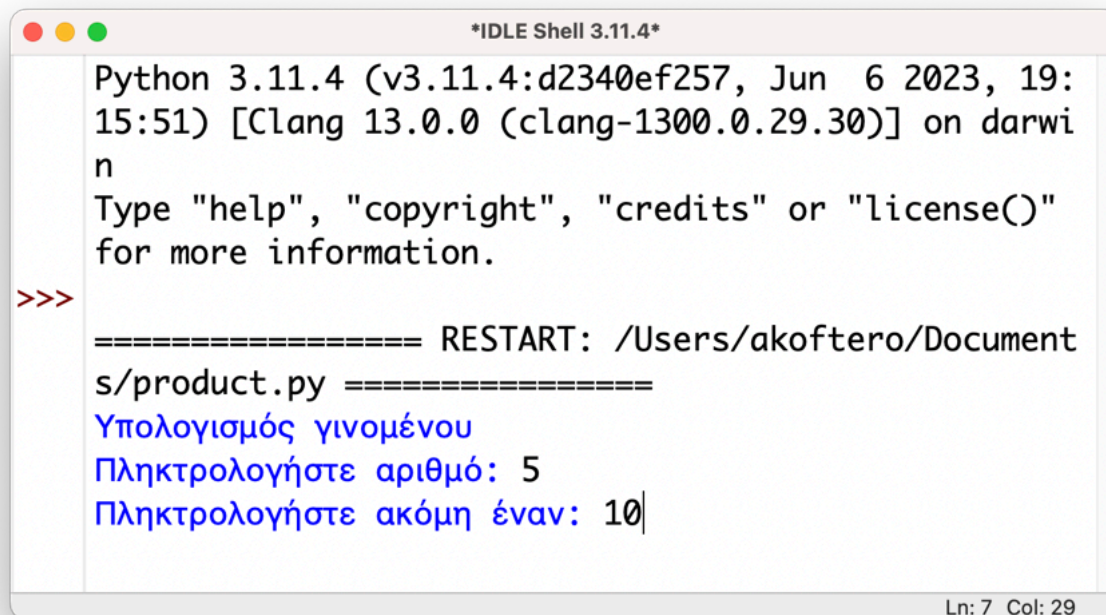


Let's look at a more "serious" mistake. We will create a simple program to calculate the product of two numbers:

```
print("Product calculation")
number1=input("Type a number: ")
number2=int(input("Type a second number: "))
print("The product is ", number1*number2)
```

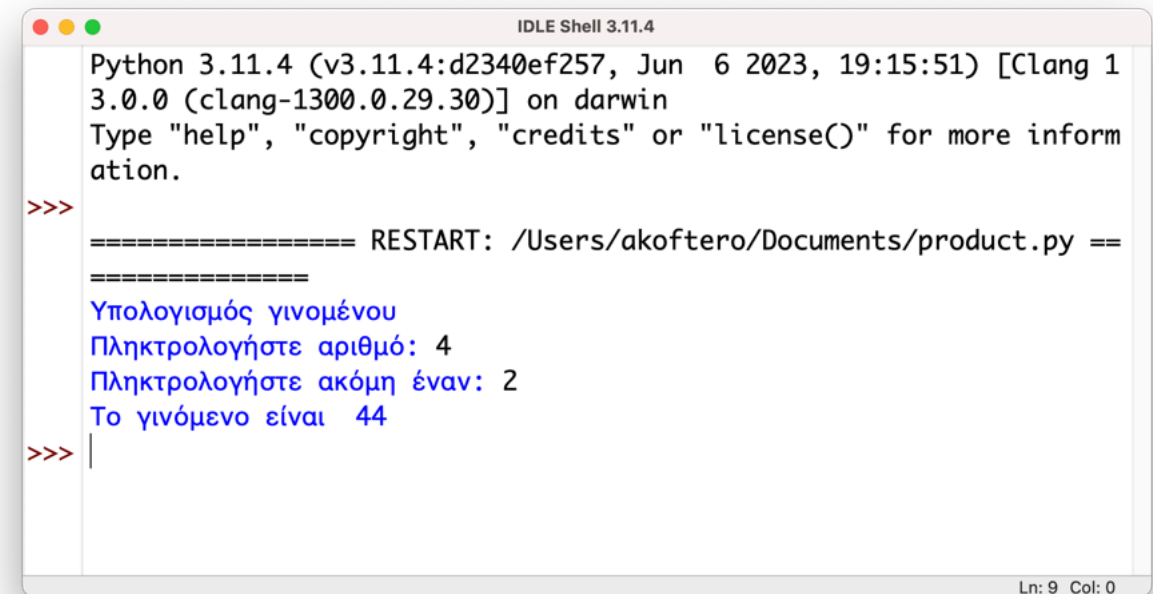
There is no "error" in the above code. During its execution, it will indeed ask us to enter 2 numbers from the keyboard. Each number will go into a variable (number1 and number2).

When we run the program, the first 3 lines will be executed normally:



```
Python 3.11.4 (v3.11.4:d2340ef257, Jun 6 2023, 19:15:51) [Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: /Users/akoftero/Documents/product.py =====
Υπολογισμός γινομένου
Πληκτρολογήστε αριθμό: 5
Πληκτρολογήστε ακόμη έναν: 10
```

We have given 2 numbers from the keyboard. When we press the ENTER key to proceed with the execution of the last command, the following result appears:



```
Python 3.11.4 (v3.11.4:d2340ef257, Jun 6 2023, 19:15:51) [Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: /Users/akoftero/Documents/product.py =====
Υπολογισμός γινομένου
Πληκτρολογήστε αριθμό: 4
Πληκτρολογήστε ακόμη έναν: 2
Το γινόμενο είναι 44
>>>
```

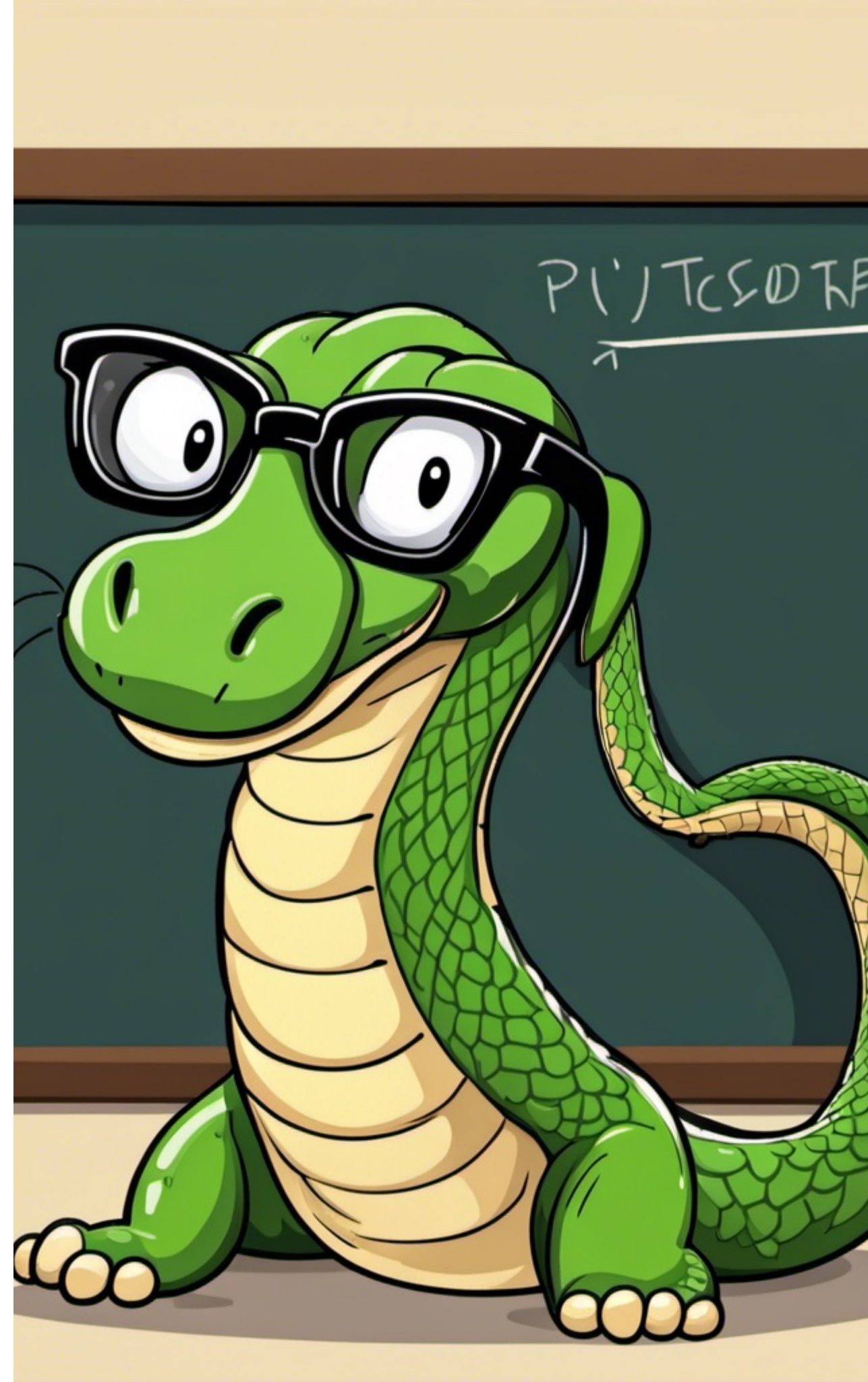
If we run it again, even with the same numbers (4 and 2), we will see how an incorrect result is obtained. The reason is simple: the first variable takes an "algebraic" value and not an integer! We should change it to:

```
number1=int(input("Type a number:")).
```

With this change, the number we will give to this variable will be recognized by our code as an integer.

What have we learned so far?

- In chapter 3 we worked with several lines of code and created a simple game.
- With the `print()` commands we can display several lines of information on the screen at the same time.
- Variables are used to store values (text, integer, decimal...) which we use in the code, usually for comparison purposes, mathematical operations, etc.
- With the `input()` command we can give a value to a variable from the keyboard.
- With the command `if...else` we can check if a condition is true (e.g. `10 > 5`).
- With `type()` we can determine the type of the variable.
- With `int(input())` we input a value as an integer.



Activities

1. What is the result of running the code below? Try writing it without running it on the computer.

```
name=input("Please write your first name")
surname=input("Please write your last name")
print("Welcome",name, surname)
```

2. What will be displayed on the screen when the code below is executed? Write it without running it on the computer.

```
print("*****")
print("*   Calculating powers   *")
print("*****")
print()
base=int(input("Please give a number"))
power=int(input("Please give the power"))
print("The",base,"on the power of",power,"equals
to",base**power)
```

3. Write a program that prompts the keyboard for a number from 1-10. Then display the multiplication table of that number.
4. Write a program to request the "code" from the keyboard. If the password is correct ("mypassword"), the message "Very correct" should be displayed. Otherwise, it will give you the message "you made a mistake" and stop.
5. Correct the code below so that the program works correctly.

@Calculation of the product between 2 numbers

```
print(Welcome to our program)
number1=input("Type the first number")
number2=input("Type the second number")
print("Their product is",number1*Number2)
```

4. Repetitions...

```
10 | print ("It takes all the running you can do,  
20 | to keep in the same place. If you want to get  
30 | somewhere else, you must run at least  
40 | twice as fast as that!")  
50 | #Alice in Wonderland
```

Repetitions

When we write programs, we often need to repeat some commands. For example, in the game "Guess the number I'm thinking", we should give 2 or even 3 chances to someone to find it.

One way is to write all the commands 3 times. Another way is, with some commands, to repeat their execution 3 (or more) times.

Loops are particularly useful because they allow us to create complex programs without having to repeat the same commands over and over (in some cases this will be necessary).





What we are going to learn:

In **Unit 4 "Repetitions..."** we will learn the advantages of using loops to execute repeated commands. It will also:

- Let's learn while() and use it in conjunction with if().
- We use variables to control the number of iterations.
- We will learn and use the for loop in conjunction with range().
- We perform iterations with a certain range of numbers.
- We perform repetitions to create shapes and to calculate the product of numbers.



The iterative Python!

Pythons are particularly stubborn animals - they can do something over and over until it's done right or until they get bored - whichever comes first!

In the previous section, we created a simple age of python game. Our game, if we give a wrong answer, it will be over! But we want to be able to continue until we give the right answer (or, at least, give 2 or 3 answers before it ends).

These commands are called "**loops**". There are different types of repeat commands that we can use. For our game, we will use the **while** command.

What we want is for the game to continue until we enter the number 10 from the keyboard.



Let's look at the command:

```
while age!=10:
```

In the command above, as long as the value we type is different from 10, the program will continue to execute the commands:

```
age=input("How many years does a python live?\n")
```

This command is necessary so that it continues to ask us to enter another number.

```
if age==10:  
    print("Correct!")  
else:  
    print("Wrong")
```

Before the while command, we should give an initial value to the variable we will use. This value should not be the same as the one it controls (eg anything other than 10).



Next we will study other examples of repetitions.

In the previous example, we saw how to return Python's age over and over again until we found the correct result. This can take hours (or even days!) if we don't set a limit.

We're going to change our code a bit so that it stops asking for a number after 3 tries.

We start with a new variable, which we will use as a "counter":

```
count=1
```

With the above variable, we will measure our efforts (and therefore, how many repetitions we will do).

```
while count<3:
```

With the above change to while, the iteration will continue as long as the value of the count variable is less than 3.

```
age=input("How many years does a python live?
\n")
print("Correct!") if age==10 else
print("Wrong!")
```

With the above commands, we get a value from the keyboard, which will be stored in the age variable. In the

next line, a check is made as to whether age has the value 10.

Here we need to add one more line: after the if check, we need to increase the value of count by 1. This is important, because we said that it will only be repeated 3 times. The code is:

```
count=count+1
```

In programming, commands are read from right to left. The above command reads as "add 1 to the value count already has, and store the new content in the same variable (count).



The change in the count value can also be written differently:

```
count+=1
```

It seems a bit strange, but we read it like this: "to the value of count, you should add 1".

But here we have a problem: if we run our program, it terminates in two attempts.

On the previous page, we saw the repetition of our commands. But instead of 3 repetitions, there were only 2! To solve this "mystery".

```
count=1
while count<3:
    count=count+1
```

Initially count has the value 1

while checks if count is less than 3. Since it is (count=1), it moves on to the next statement (count=count+1). Now the count is 2. This is the **first iteration**.

The while again checks if count is less than 3. As we saw, count=2, so it continues executing the statements. The next command tells count to increase its value by 1. Since count was 2, it will now be 3 (count=2+1). This is the **second iteration**.

Count has the value 3 now. In the while check (whether the count is less than 3), we see that it is no longer valid. And it stops working on the second iteration.

For our program to work as we want (give us at least 3 attempts) we should make the following change:

```
count=0 (start counting from 0 not from 1)
while count<3:
    count=count+1
```

This is a solution. Another solution is to keep 1 as the initial value of count and simply increment the value in the check.

```
count=1
while count<4:
    count=count+1
```

Except... there's one more problem with our code (you might have guessed it already).

On the next page we will make one more change to the code so that the program works properly.



Breaking Good!

In the code above, the loop continues for 3 tries. But even if we get the age right the first time, it will continue for 2 more!

This is easily fixed with the break statement and some changes to the if condition.

```
count=1 #initial value of count
while count<4: #repetitions start
    age=int(input("How many years does a python
live?"))
    if age==10:
        print("Correct answer!")
        break #σταματά η εκτέλεση της while
    else:
        print("Wrong answer!")
    count=count+1 #repetitions end
```



```
myfirstpython.py - /Users/akoftero/Documents/myfirstpython.py (3.11.4)
print("*****")
print("Welcome to the program 'Find my age'")
print("*****")
print()
print()
#With the following commands, we ask for a number
#to guess the age a python can reach

count=0 #counts number of attempts
age=1 #initial value of age
while count<3:
    age=int(input("How many years does a python live?"))#we ask for an integer
    print(age)
    if age==10:
        print("You have found the correct age!")
        break
    else:
        print("Unfortunately you were wrong!")

count=count+1 #increases the count by 1
```

The image shows the entire code we wrote! We have created a simple game, with 18 lines of code, in which we used variables, repetition (while), comments and a condition to check the value of the variable (If...else).

In the image above, we see that we have left a space in rows 8 and 19. These spaces are ignored by Python when executing the commands. They are useful for making the code readable.



One, Two, Three, For!

For repetitions, we can also use the For command.

We will create a new game, in which you will have to guess a number that the computer "thinks"! We will use the loop, as well as the range() function.

range() is an important function. It helps us to use a range of values (from one number to another).

Let's look at the commands:

```
for counter in range(3):  
    print(counter)
```

Let's look at the commands:

Counter is a variable. With the range(3) function, the variable takes values from 0 (which is the initial value) to 2. The number 3 tells us, that is, to take 3 numbers in a row, BUT the first one is 0 (so the numbers are 0, 1 and 2).

The for statement will execute the program 3 times. On the first run, it will print the number 0 on the screen, on the second run, the number 1, and on the third run, the number 2.

0
1
2

Let's look at an example of using for to display a multiplication table:

```
for counter in range(1, 13):
```

In the line above, we set the range to be from 1 to 13. If we put range(13), then it would start from 0 to 12. We want it to start from 1 to 12. We also add the following command :

```
print(8*counter) #Array of 8 will appear
```



```
IDLE Shell 3.11.4  
8  
16  
24  
32  
40  
48  
56  
64  
72  
80  
88  
96  
Ln: 20 Col: 0
```


While... or...

Next we will learn how to repeat within another iteration.

This is called a **"nested loop"** or **"nesting"** of repetition.

Let's look at using **while()** to create a program to calculate all the multiplication tables, from 1-12:

```
print("*****")
print("* Multiplication tables *")
print("*****")
print()
```

With the above commands we give information on the screen about what the program will do. Then we will ask the user to select the multiplication table:

```
myArray=int(input("Select an array from 1-12"))
```

We want to limit the selection from 1-12. We will use while() to check if the number given to us by the keyboard is from 1-12. We want a double check: if they give us a number less than 1 (eg 1) it should output a message that we have to give a bigger one. If they give us a number greater than 12, then it should output a message that we must give a smaller number.

If we used the "if" by itself, it would do the check, but it would only be done once (no repetition). Let's look at an example with while:

```
while myArray<1:
```

The above line constantly checks if the value we give to the variable "myArray" is less than 1. If it finds that the value we give is less than 1, then it displays the command below:

```
pinakas=int(input("Please give a number  
larger than or equal to 1"))
```

The above is very important, because it will keep showing it until we give a number greater than 1! It also displays the message about what number it expects us to give it.

It is very important to give information about what information we want from the keyboard. So it is important to inform (above) that we need to give a number greater than 1 to continue.



The commands we have given so far tell us what the program does (Multiplication Tables) and asks us to select the table we want and stores it in the variable "pinakas" as an integer (int). Then it checks if the number we gave is less than 1 (because we want it to choose from the array of 1 to 12).

But, it should also check if the number we give is greater than 12! If we type, for example, 14, it should tell us that the number must be equal to or less than 12 and ask us to enter another number.

```
while myArray>12:  
    myArray=int(input("Please give a number  
equal or bigger than 12"))
```

With the above command, our program constantly checks if the number we entered is greater than 12. If it is greater, then it displays a message asking us to enter a number equal to or less than 12.

But, there is a problem... If we run the code, we will see that the check is done, indeed, but only the first time! And it doesn't check the value we have given at the same time. Here we should make a change to the code: use while, but

tell it to check whether the number is less than 1 or greater than 12.

```
while myArray<1 or pinakas>12:
```

With the above command, it repeats all those included in the loop if the number is less than 1 or greater than 12. But we want it to give us a message so we know what we did wrong and what number to give. Thus we have the following conditions:

```
if myArray<1:  
    myArray=int(input("Please type a number  
equal or greater than 1"))
```

With the above commands, if the number we typed is less than 1, then it will prompt us to give a number equal to or greater than 1. Then we will check if the number we typed is greater than 12.

```
else:  
    myArray=int(input("Please give a number equal  
or greater than 12"))
```

So far, our code asks for a number from the keyboard and checks if it's between 1 and 12. Next, we'll display the multiplication table of the number we selected. We will use a for loop:

```
for multiplier in range(1,13):  
    print(myArray,"X",multiplier,"  
=",myArray*multiplier)
```

```
multiplication.py - /Users/akoftero/Documents/multiplication.py (3.11.4)  
print("*****")  
print("* Multiplication tables *")  
print("*****")  
print()  
myArray=int(input("Please choose a table between 1-12: "))  
  
while myArray<1 or myArray>12:  
    if myArray<1:  
        myArray=int(input("Please give a number greater than 1"))  
    else:  
        myArray=int(input("Please give a number smaller than 12:"))  
for multiplier in range(1,13):  
    print(myArray, "X", multiplier, "=", myArray*multiplier)
```

```
IDLE Shell 3.11.4  
* Multiplication tables *  
*****  
  
Please choose a table between 1-12: 1  
1 X 1 = 1  
1 X 2 = 2  
1 X 3 = 3  
1 X 4 = 4  
1 X 5 = 5  
1 X 6 = 6  
1 X 7 = 7  
1 X 8 = 8  
1 X 9 = 9  
1 X 10 = 10  
1 X 11 = 11  
1 X 12 = 12  
>>>
```

The image above shows the result of executing our code. We have created a fairly complex program, with a while loop, a for loop and if conditions. Our completed code appears below:

Our program, although quite complex, lacks one basic function: to calculate another table, we have to run it from the beginning. Next, we'll see how we modify the code so that it continues with a new multiplication table unless we break it! To do this we need to place iteration inside iteration. This is called a nested loop.



"Nested" Repetition

next we will learn how to iterate within another iteration. As we mentioned on the previous page, this is called a "nested loop" or "nesting" of repetition.

In the previous program, we requested a value from the keyboard and then the multiplication table was displayed. But when the table appeared, the program would stop.

Next we will create a program that calculates the matrix of a number from 1 to 12, and then it will ask us if we want to continue with another number.

In other words, we should have **two repetitions**, one inside the other:

- One iteration will calculate all multiples of our number, from 1 to 12.
- This iteration will be inside another iteration that will start the process again, with a new number.



For our example, we will make a simpler program than the previous one.

```
print("*****")
print("* Multiplication Tables *")
print("*****")
print()
```

`x="y"` #we give initial value to x

The variable x is necessary, because with it we will check whether or not the execution of the program will continue.

`while x == "y":` #checks if x has the value y

The first iteration starts with the above command. It will repeat all the commands that follow, until we give the letter 'y' from the keyboard.

Next, we give the rest of the commands to calculate the multiplication table. Inside the while we will have a for loop.

```
pinakas=int(input("Give a number: "))
for multiplier in range(1,13)
    print(multiplier*pinakas)
```

Once the for loop completes, a message to continue is displayed:

```
x=input("Press y and ENTER to continue, any  
other key to exit")
```

With the above command, the repetition starts again if we type "y" (and press ENTER) or the program is terminated. The last remaining command is print(), which will display a message when we terminate the program.

```
print("Thank you!")
```

The entire code appears below:

```
print("*****")  
print("* Multiplication Tables *")  
print("*****")  
print()
```

```
x="y" #we give initial value to x
```

```
while x == "y": #checks if x has the value y  
    pinakas=int(input("Give a number: "))  
    for multiplier in range(1,13)  
        print(multiplier*pinakas)  
    x=input("Press y and ENTER to continue,  
any other key  
to exit")  
print("Thank you!")
```

With these commands, the program will run continuously and calculate the multiplication table of the number we



gave. By pressing the "y" key (and ENTER), the process will start from the beginning, until we press any other key except "y".

Our code has a small problem: if we press the "y" key and we have Caps Lock enabled (or we hold shift), then it will be typed as capital Y. Because "y" and "Y" are considered to be completely different, the answer will not be recognised!



In the **Activities**, at the end of this chapter, there is a related exercise to solve this problem.

What have we learned so far?

- In chapter 4 we learned about while and for loops.
- With the while loop, a series of statements is repeated as long as a relation holds true (eg $x=10$).
- With the for loop, we check whether a certain number of iterations have been done.
- `range(0,12)` gives us a certain range of numbers.
- In an iteration (loop) we can have another iteration, inside the first one. This is called "nesting" or nested loop.



Activities

1. What is the result of running the code below? Try writing it without running it on the computer.

```
x=int(input("Please give a number:"))
for count in range(0,12)
print("The array of ",x,"is",x*count)
```

2. What is the result when the following code is executed;

```
print("*****")
print("*   Power calculation   *")
print("*****")
print()

x="y"
while x == "y":
    base=int(input("Please give a number:"))
    power=int(input("Please give the power:"))
    print(base**power)
    x=input("Press y and then ENTER")
print("Thank you!")
```

3. Write a program that asks for a distance in meters and then returns you the centimeters (eg 1 meter = 100 centimeters).
4. Write a program to add 3 numbers. It should ask you for 3 different numbers and repeat until you press an exit button (eg "y").
5. Rewrite the example program of the "Nested" iteration section. The code should detect whether we pressed "y" or "Y" and recognize them as the same option.
6. Correct the code below:

```
print("*   Calculation of difference   *")
x=int(input("Give a number")
y=int(input("Give another number")
while answer=="y
    print(x-y)
    answer=input("Press y to continue)
```

5. Functions

```
10 | print ("Have I gone mad?")  
20 | print ("Am afraid so. You're totally bonkers!")  
30 | But let me tell you something.  
40 | All great people are!")  
50 | #Alice in Wonderland
```

Functions

This word is definitely unusual... “functions”... something you will come across in Math in older classes. However, functions (as they are called in English) are very useful, as we will see in the following pages.

Functions we have already used in the examples of the previous chapters.

`print()` is a function, which displays the content of the parenthesis on our screen (or other devices).

`input()` is another function, which asks us to input data (number, text) from the keyboard.

In the next few pages we'll learn more about functions, and create our own.





What we are going to learn:

In **Unit 5 "Functions"** we will get to know the structure of a function. It will also:

- Let's create our own functions.
- Embed conditions in a function.
- Use loops in a function.
- Let's use functions to create shapes on our screen.



The game of actions

Pythons, as we know, are very good at Maths. They can remember in every detail even how many grasses and branches they encountered on their way. For us it is not so sure, so we will write a program to help us with integer operations.

```
myfirstfunction.py - /Users/akoftero/Documents/Development Projects/myfirstfunction.py (3.11.4)
#Program to learn the use of functions
#we will give two numbers and Python will calculate
#their sum or product

print("*****")
print("  Welcome to the automatic calculator  ")
print("*****")

number1=int(input("Please type the first number"))
number2=int(input("Please type the second number"))
```

In the code above, we explain its basic function with comments. Then we display informational messages on the screen with `print()` and then, with the keyboard, we give 2 numbers:

- one will be the value of the variable `number1`, and
- the other will be the value of the variable `number2`.

Next, we will perform the operations with these two numbers.

```
print("The sum of the two numbers is",number1+number2)
print("The product of the two numbers
is",number1*number2)
```

When we run the program, we will see the following:

```
IDLE Shell 3.11.4
== RESTART: /Users/akoftero/Documents/Development Project
s/myfirstfunction.py ==
*****
  Welcome to the automatic calculator *
*****
Please type the first number: 5
Please type the second number: 4
The sum of the two numbers is 9
The product of the two numbers is 20
>>>
```

Up to this point, our code is no different from what we've seen so far. Next, we'll modify the commands to create our first function.



Our first function!

Some commands can be **grouped** together to do a **specific task**. Thus, we have some commands that are only concerned with calculating the product, while others are the sum of two numbers.

We will create a group of commands that we will call "mymultiplications" and a group of commands that we will call "myadditions". The names don't necessarily mean anything, as long as we follow the right rules (as we saw on previous pages).

The groups of commands we create are called **functions**.

To create our first function, we first start with the `def` command, followed by its name (`mymultiplications`) and parentheses:

```
def mymultiplications():  
    print("The product is",number1*number2)
```

All commands below the function (with spacing from the beginning of the line) are part of the function.

If we try to run the program, we will see that nothing appears. This does not mean that there is a problem with our code. On the contrary! We will then see how we "call" the functions!

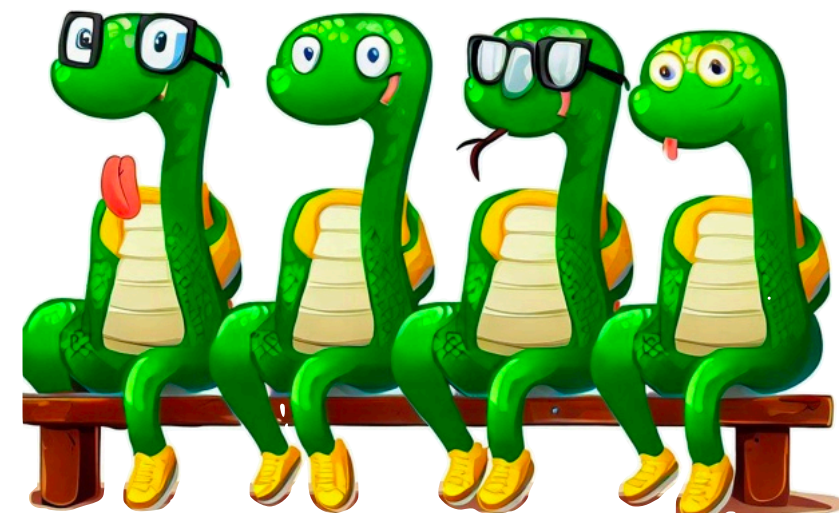
Functions are like substitute players - they know their instructions very well, but they wait patiently for us to call them! If we don't call them, they are there somewhere, but they don't do anything.



To call a function, we type its name:

`mymultiplications()`

The commands contained in the function will be executed and the result will be shown on the screen.



Choosing between functions...

Next, we'll use if conditions to choose between functions to execute. We will give two numbers from the keyboard. Next, we will choose whether to do addition or multiplication.

```
num1=int(input("Please give a number"))
num2=int(input("Please give a second number"))
```

With the above two commands, we enter two numbers from the keyboard.

```
def mymultiplication():
    print("The product is",num1*num2)
```

With the above function, we calculate the product of the two numbers.

```
def sum():
    print("The summary is",num1+num2)
```

With the sum() function we calculate the sum of two numbers.

Next, we'll give options to choose whether we want to add or subtract:

```
choice=input("Please press: a to add, b to multiply")
```

With the above command, type either a or b to select the type of action to perform.

```
if choice=="a":
    sum()
if choice=="b":
    mymultiplication()
```

With the above commands, we can choose whether to call the condition with which the sum will be calculated or the condition with which the product of the two numbers will be calculated.

With the functions we will create complex geometric shapes, with the help of the "turtle" module for Python. We will learn more about functions and geometric shapes in **Part C, "Turtles and Pythons"**.



What have we learned so far?

- In **Chapter 5** we learned about functions.
- We created our own function with the `def` command.
- We called the function we created inside our code.
- We combined iterations with our functions.



Activities

1. Try, within a function, calling the same. What do you think will happen? Try to explain it.

```
def mymultiplications():  
    print("Hello")  
    mymultiplications()
```

```
mymultiplications()
```

2. Find the mistakes in the meeting below and correct it. What will be displayed on the screen when executed?

```
def division()  
    print("The quotient is", 25/5)  
    divisions()
```

```
divisions()
```

3. Without running the code below, write the result it will have on your screen:

```
print("*****")  
print("*   Calculate the product   *")  
print("*****")  
print()
```

```
#---Function-----
```

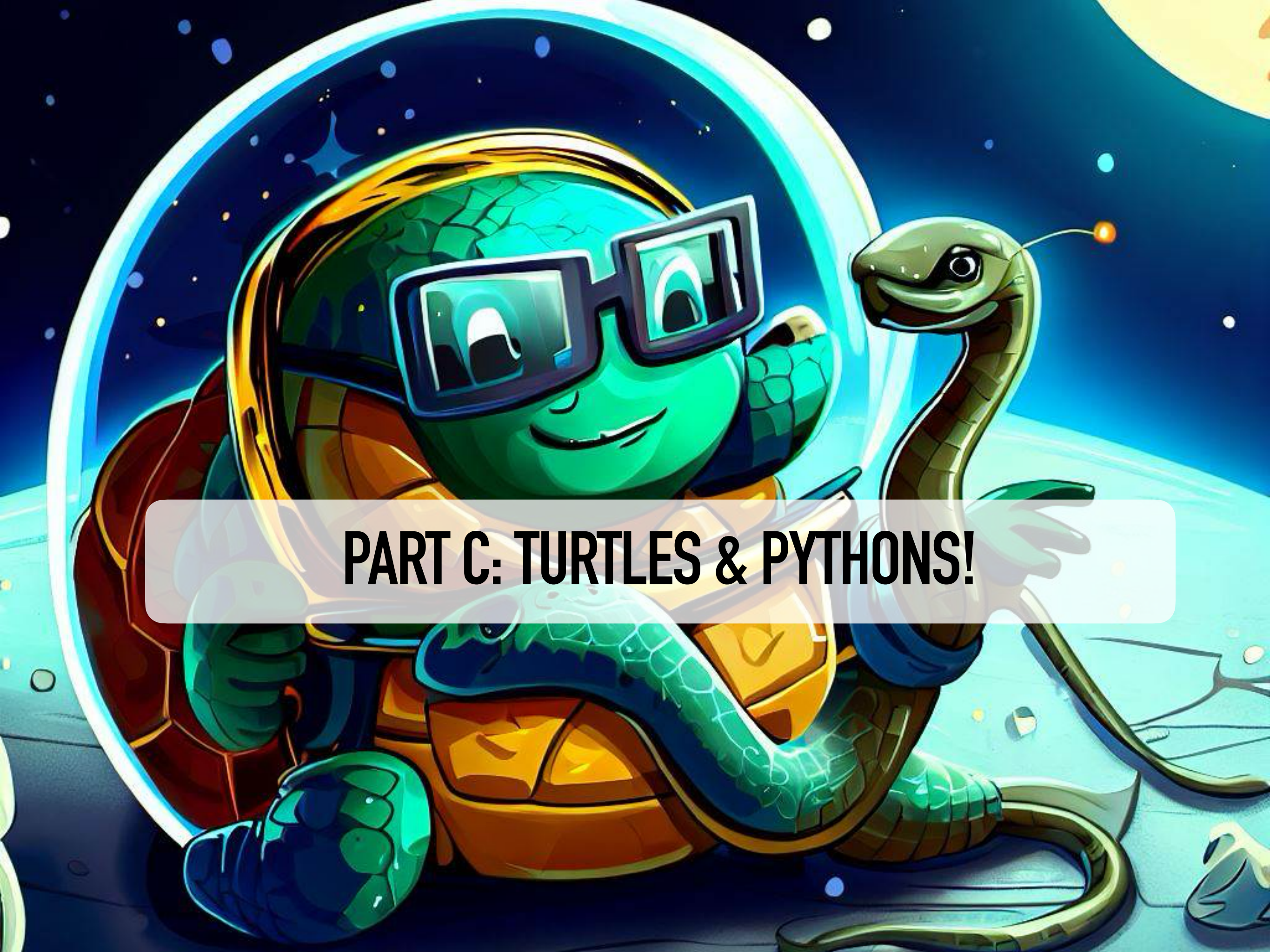
```
def mymultiplications():  
    print("The product is", number1*number2)
```

```
#---End of Fuction-----
```

```
number1=int(input("Give a number"))  
number2=int(input("Give another number"))
```

```
mymultiplications() #calling the function
```

4. Write a program that prompts you for a number and then chooses whether to (a) display its multiplication table (up to 12) or display the number to the second power (the product of itself).

A vibrant, cartoon-style illustration featuring a green turtle with large, black-rimmed glasses. The turtle is smiling and holding a small, green snake with its mouth open. The background is a deep blue space scene with a large, glowing planet in the upper left and several stars. The overall style is playful and whimsical.

PART C: TURTLES & PYTHONS!

TURTLE GRAPHICS!

As strange as it sounds, *Pythons can draw*.

Ok, maybe not the real ones - and we don't want you to get too close to see for yourself - but the digital ones for sure. In fact, they borrow the "tools" from their friends the turtles.

In Part 3 of this book we will deal with the commands that allow Python to create shapes. Specifically, we'll use the joint (weird word, right?) with the commands that allow you to create line shapes.

Next, we will use these commands to create geometric shapes.





What we are going to learn:

Through **Part C "Turtles and Pythons"** we will have the opportunity to:

- Know and introduce the turtle joint.
- Know the basic commands for creating shapes.
- Create our own shapes using repetitions.
- Understand the way in which our turtle "moves" on the screen.
- Use functions to create complex geometric shapes.



Python Modules

The possibilities of Python are almost limitless. This is (also) due to the fact that we can use **modules** to give many new features to the programs we create, without having to write complex code. Ok, we're definitely going to write a lot of code, but let's keep it as simple as possible.

The turtle module

A useful module is **turtle**. It is based on the LOGO programming environment created in 1967 to teach math to children in the US (using a robot turtle).

If you have installed Python using the instructions in this book, then you also have the turtle module on your computer.

The logic of LOGO - and in our examples, of Python with the additional commands - is simple: we give commands (FORWARD, LEFT, RIGHT, etc.) to a "turtle" on our screen. The "turtle" moves on the screen and creates designs. These (geometric) designs can have a different shape (border), line color, etc. The possibilities offered by the turtle module are (almost) limitless, and we will get to know quite a few of them.

The first thing we need to do is to insert the turtle joint:

```
from turtle import *
```

Next we type our first command to create a schema:

```
forward(100)
```

The above will create a line with a length of 100 pixels.

We'll get to know more about pixels and shape commands next.



Pixels!

The screen of every digital device, from the computer to our mobile phone and smart watch, consists of very small squares - the pixels (Pixels, from the words Picture Elements).



Modern screens have so many lines and columns of pixels (the "resolution" we say...) that we cannot easily distinguish them. Of course, in earlier times when screens had much fewer pixels, the squares of the screen looked very clear. And the graphics were much more 'square'.

When we give the command `forward(100)`, we ask our "turtle" to move in the direction it is "looking" by 100 pixels.

Each screen has a different resolution. And it doesn't always depend on its size. For example, an older computer with a 17" screen may have a much lower resolution than a modern laptop with a 14" screen.

If your tablet or laptop has an **HD** screen, it means that it consists of 1920 **columns** and 1080 **rows** of **pixels**. If we want to know how many pixels that is in total, we can use our calculator, or pencil and paper, or Python itself.



```
>>> 1920*1080  
2073600
```


Colors & Lines!

What about the colors? Shapes are good, but we also want some color in our classrooms, in our neighborhood, in our books, on our screens!

Before the drawing commands, we can choose a color (we'll also see how we change colors along the way).

```
from turtle import *  
color("blue") #color turns to blue  
forward(100)
```

In the color() command we put in parentheses the color we want to use in the creation of a shape. For the above example, we create a blue line 100 pixels long.

Let's try different combinations:

```
from turtle import *  
color("blue") #color turns to blue  
forward(100)  
color("green") #color turns to green  
forward(100)
```

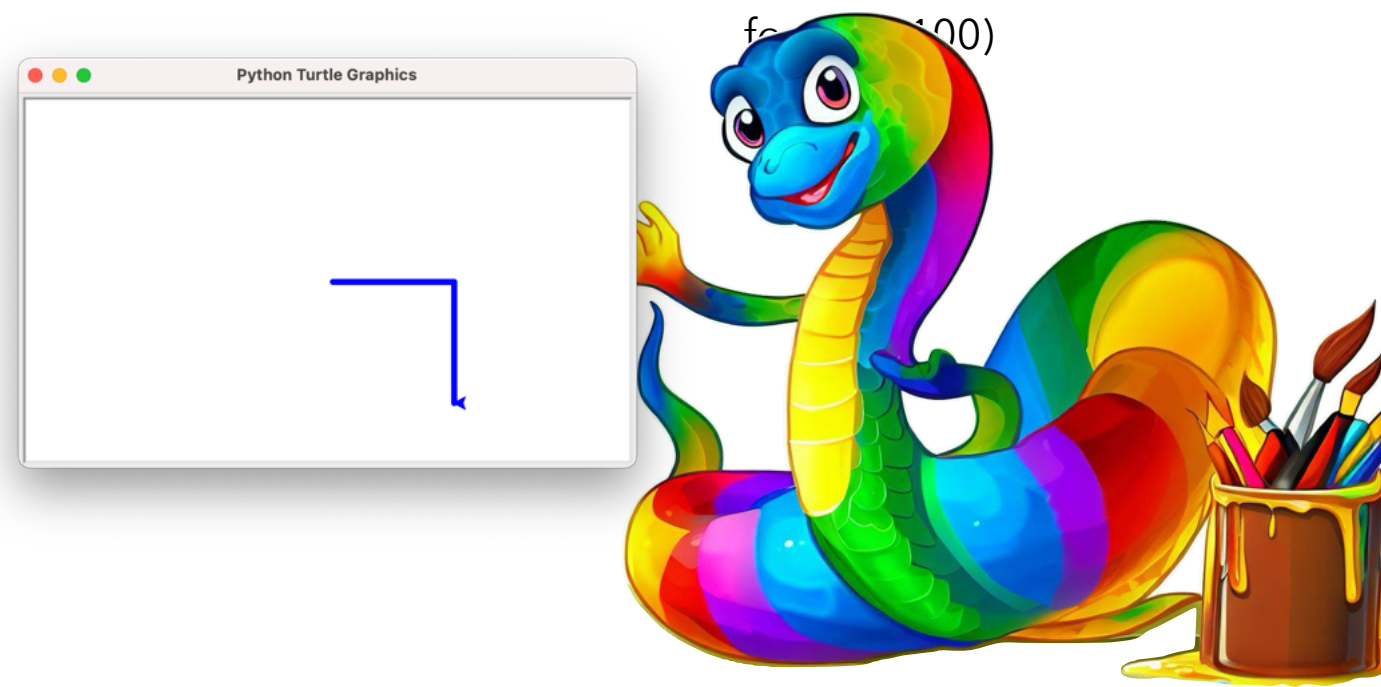
We can easily see that changing the color is a relatively simple process.

Before the command with which the shape will be created, we also choose the color. We can choose the color by its name: black, blue, red, yellow, pink, brown, gold, maroon, etc.



To change the width of the line, we use the width() command with a number:

```
from turtle import *  
color("blue") #color turns to blue  
width(5)      #width of the line turns to 5  
forward(100)  
right(90)
```

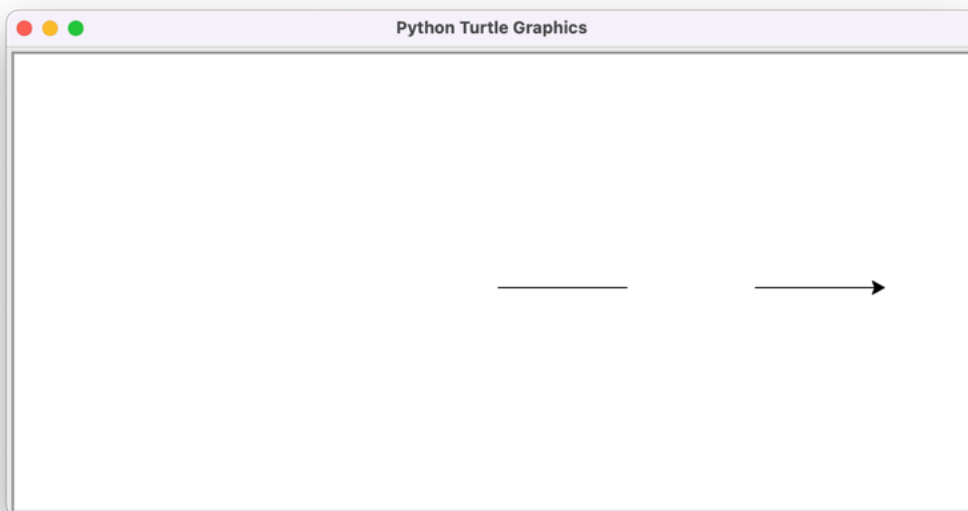


Pennies up and down...

If we want to create a line 200px long, we will give the forward(200) command. However, if we want to create a series of lines with spaces between them, then we will have to teach the turtle to "lift" the pen as it goes.

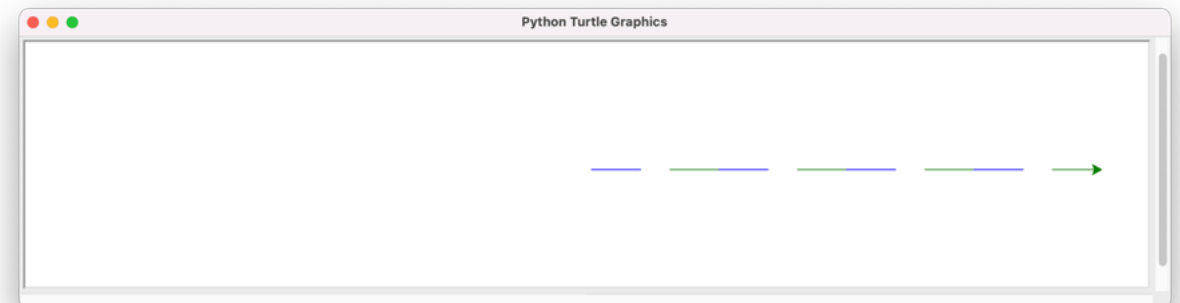
With penup(), we tell our turtle to "pick up" the pen. Immediately after that follows the movement command, so that it moves to another position. If we want it to start writing again, then we give the pendown() command.

```
from turtle import *  
forward(100) #move 100px  
penup() #pen is up,so it can't write!  
forward(100) #move 100px  
pendown() #pen is down, so it can write!  
forward(100)
```



Let's add some color (and repetition) to the code:

```
from turtle import *  
for counter in range(0, 4):  
    color("blue")  
    forward(50)  
    penup()  
    color("green")  
    forward(30)  
    pendown()  
    forward(50)
```

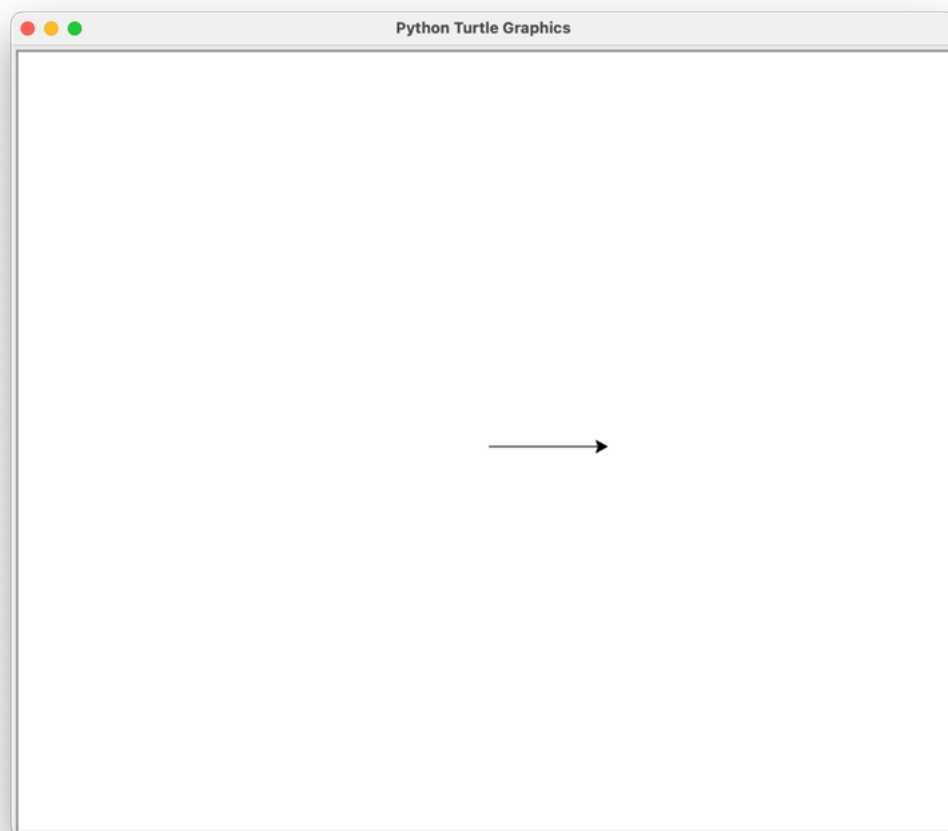


Pythons & Geometry

With the help of Python we can create all geometric shapes! Let's see examples:

```
from turtle import *  
forward(100)
```

We run the program with Run and our first line appears. Because our turtle (that's what we'll call the arrow) "looks" to the right, our line is formed in that direction.



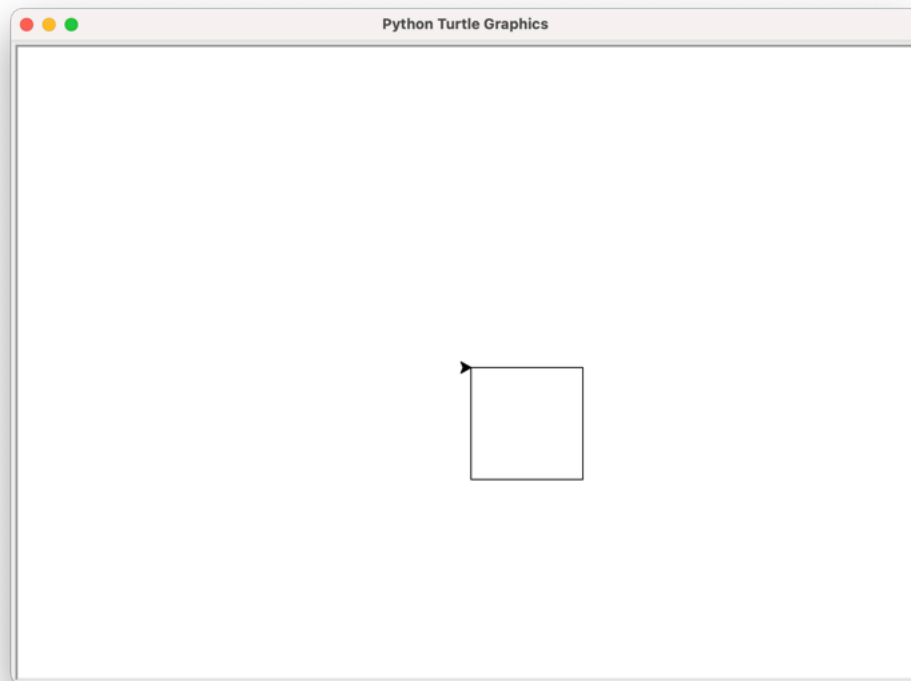
Let's create a square: we'll start with two basic commands: one to move forward (right, as shown by the turtle) and the other to turn 90 degrees to the right.

```
from turtle import *  
forward(100)  
right(90)
```

In forward(100), the number in parentheses indicates the "steps" it will take (in pixels). In the right(90) command, the number in parentheses is the degrees it will turn (to the right).

```
from turtle import *  
forward(100)  
right(90)  
forward(100)  
right(90)  
forward(100)  
right(90)  
forward(100)  
right(90)
```

On the next page we see the result of executing the above code.



Our turtle slowly creates the shape, as we see in the image above.

The code we saw on the previous page creates the square in the image above. It is important to be able to identify parts of the code that we could change and/or improve. For example, we notice that we repeat the commands `forward(100)` and `right(90)` 4 times.

We'll use `while()` to simplify our code: it should loop 4 times. That is why it is necessary to use a variable `x` that will count the repetitions.

```
*pythongeometry.py - /Users/akoftero/Documents/pythongeometry.py (3.11.4)*  
from turtle import *  
x=0  
while x<4: #ελέγχει αν το x είναι μικρότερο του 4  
    forward(100)  
    right(90)  
    x+=1 #θα μπορούσαμε να γράφαμε και x=x+1  
Ln: 3 Col: 49
```

If we run the above program, we will see that it creates the square of the image on the left. The orders have simply been reduced.

In code, it's important to identify the elements that we can simplify. It is not always necessary (or even useful) to have repetitions. But, it is very important, when writing code, to identify "patterns" that we could simplify.

The square is a special case. It would definitely be harder with a rectangular shape.

Repeat with For...

In addition to while(), we can also use the for command for repetition, as we saw in the previous chapter.

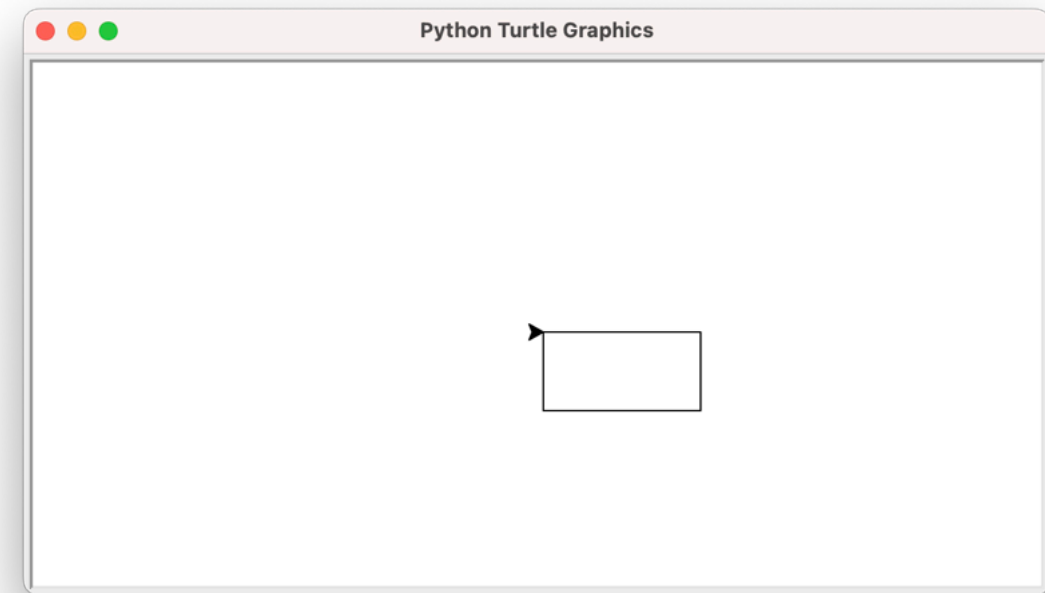
```
from turtle import *  
for x in range(0,4):  
    forward(100)  
    right(90)
```

The for command starts with a variable (here we used the variable x) which takes values from 0 to 4 (range).

It then executes the commands inside the iteration structure as many times as the range.

We'll use for to create a rectangular shape 100 pixels long and 50 pixels wide. In the rectangle, because one side is different from the other, we cannot repeat the same pair of commands 4 times. We will repeat 2 times, two pairs:

```
for x in range(0,2):  
    forward(100)  
    right(90)  
    forward(50)  
    right(90)
```



In almost every shape we can spot patterns for using repeats. Many times we may find it difficult to identify the pattern from the beginning. We can write - even on paper - the commands that our program will execute, so that we can find the pattern (if it exists) and use repetition.

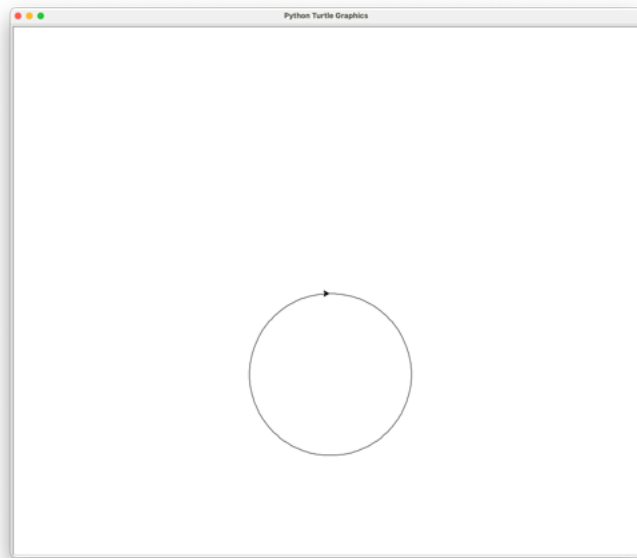


Running in circles

We will create a circle. Necessarily we should use repetition.

```
from turtle import *  
for x in range(0,180):  
    forward(5)  
    right(2)
```

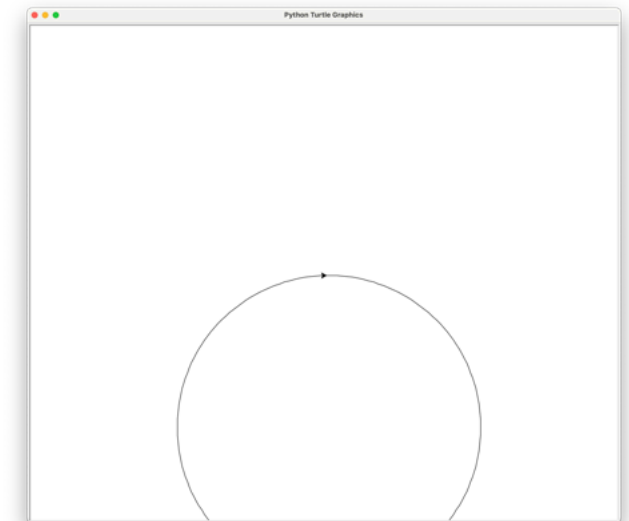
A circle has 360 degrees. In the range we gave a maximum value of 180. In the turn, we gave the command to go right 2 degrees. To close the circle, the product of the value in the range (180) with the value in the right (or left) must equal 360. ($180 \times 2 = 360$).



What if we change the forward? From 5 we double it to 10.

```
from turtle import *  
for x in range(0,180):  
    forward(10) #increased from 5  
    right(2)
```

Our circle has doubled (circumference). In a similar way we can play with the rest of the parameters (eg the range to be 90 and the turn to be 4 degrees). We can also make a left turn instead of a right turn.



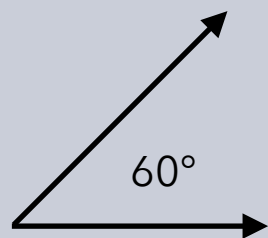
To create a circle, we can use the -you guessed it- the function named `circle()`. The number in the brackets is the size of the radius.

```
from turtle import *  
circle(10)
```

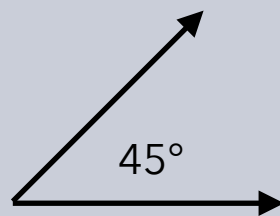


Type of angles

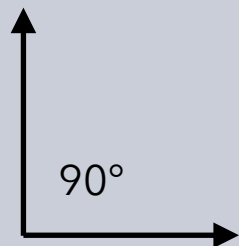
The opening of the corners is measured in degrees. Angles are measured with a special tool, the protractor. Depending on their opening, we divide them into different groups. For example, angles with an opening greater than 0° and less than 90° are called acute (images below).



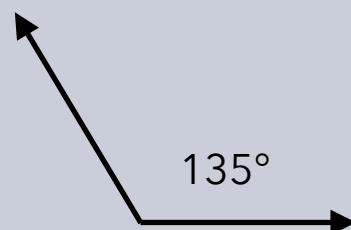
Acute angle



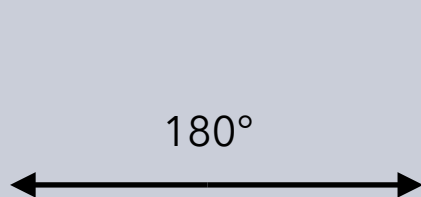
Acute angle



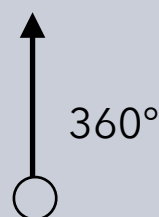
Right angle



Obtuse angle



Straight angle

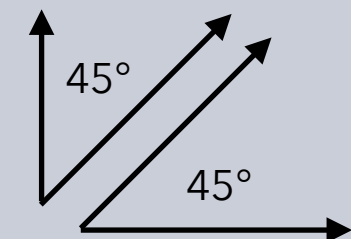
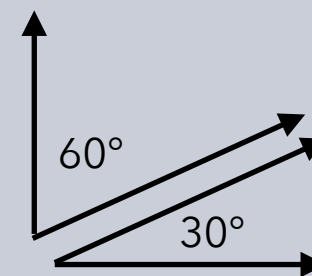


Complete angle

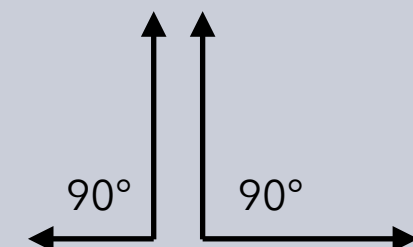
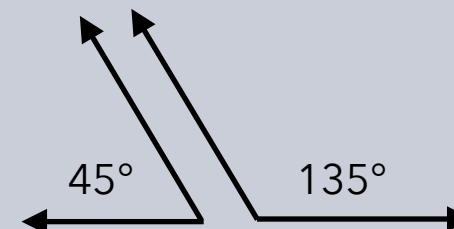
Supplementary & Complementary angles

A 90° angle is called a "right angle." A 180° angle is called a "straight angle." If you have an angle of 30° and you want to combine it with another angle to make a total of 90° , the other angle must have a measure of 60° .

These two angles, whose measures add up to 90° (making a right angle), are called "**supplementary** angles."



Two angles whose sum is 180° (a straight angle) are called **complementary** angles.



Create a triangle

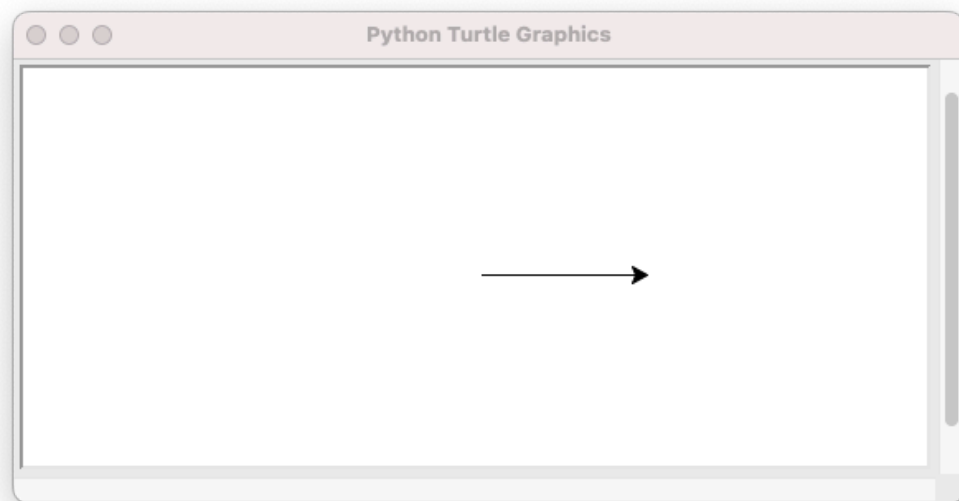
To create triangles with Python, we should have an understanding of the types of angles and especially supplementary and non-supplementary angles.

Let's look at the following example: we want to create an equilateral triangle, with the length of each side 100px (pixels). An equilateral triangle has all its angles equal (60°).

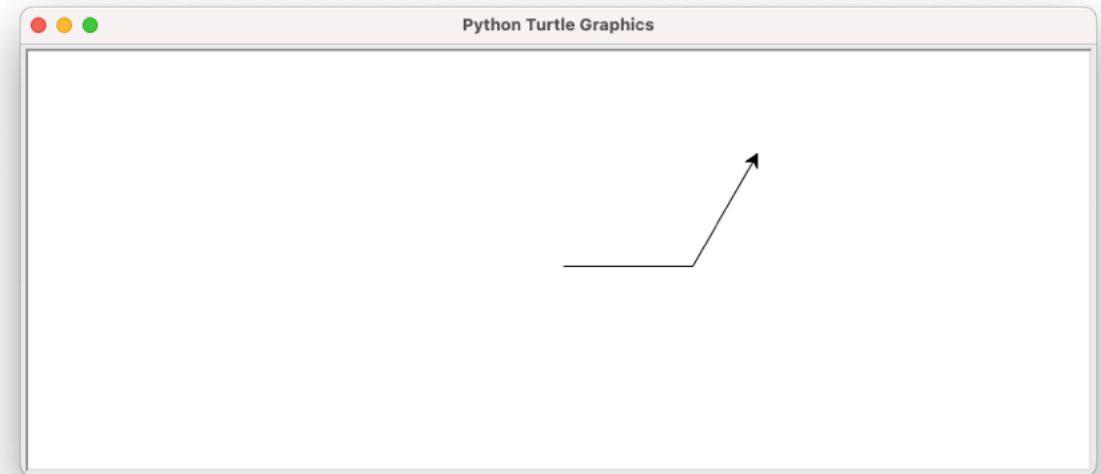
With the command below, we create the base (lower side) of the triangle:

```
from turtle import *  
forward(100)
```

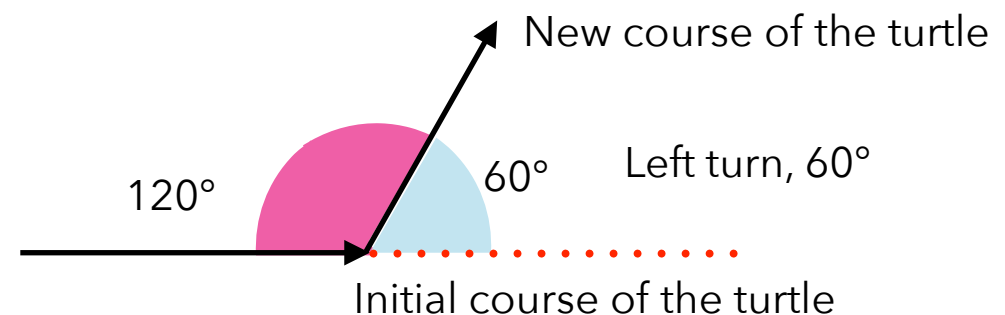
Since our turtle always faces right, we'll see this appear:



As we said at the beginning, in an equilateral triangle, each angle has a span of 60° . If we try issuing the command `left(60)` and then `forward(100)`, we will see the following:



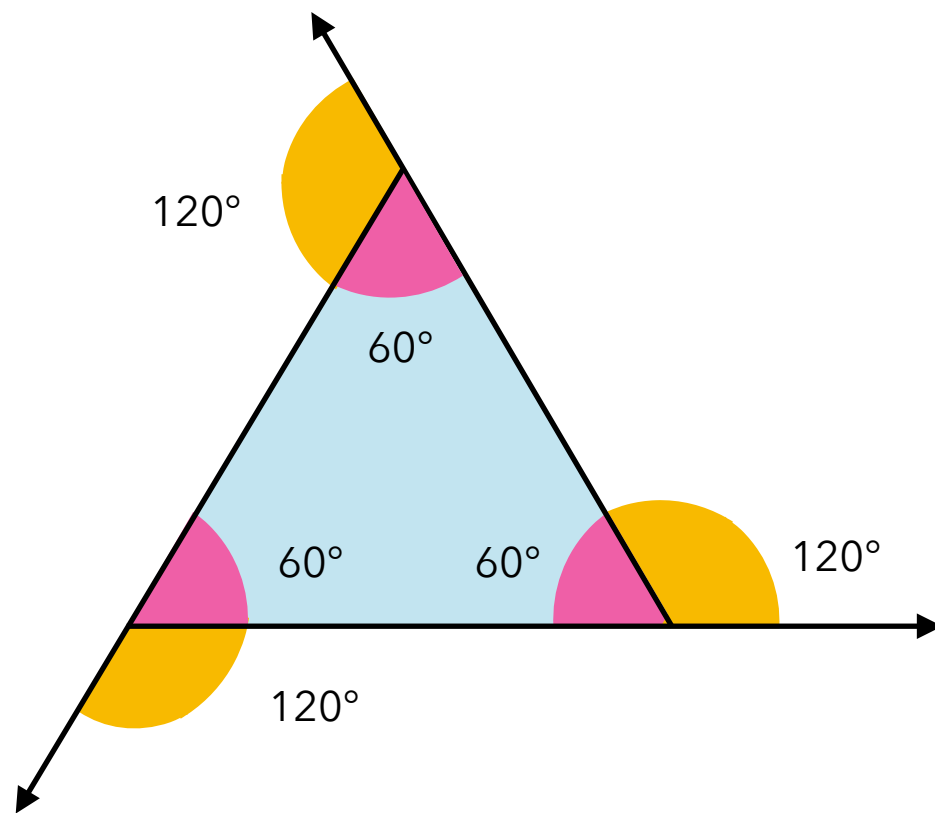
Our angle is completely wrong (or, to be precise, not the one we wanted). And it's not even 60° , it's 120° . The reason is simple: Our turtle was "looking" to the right. When we told her to turn 60° left, she did! In fact, two angles (complementary) were created.



In the example on the previous page, we see the error in the turn - our turtle is not "smart" enough to understand that we want to construct a triangle.

The "secret" is to always think with complementary (and in some cases complementary) angles: "How many degrees must the turtle turn, so that the angle that will be formed internally will be the one we expect?"

In the case of our triangle, we want the angle (inside) to be 60° , so we should turn left 120° , which is its supplementary angle.

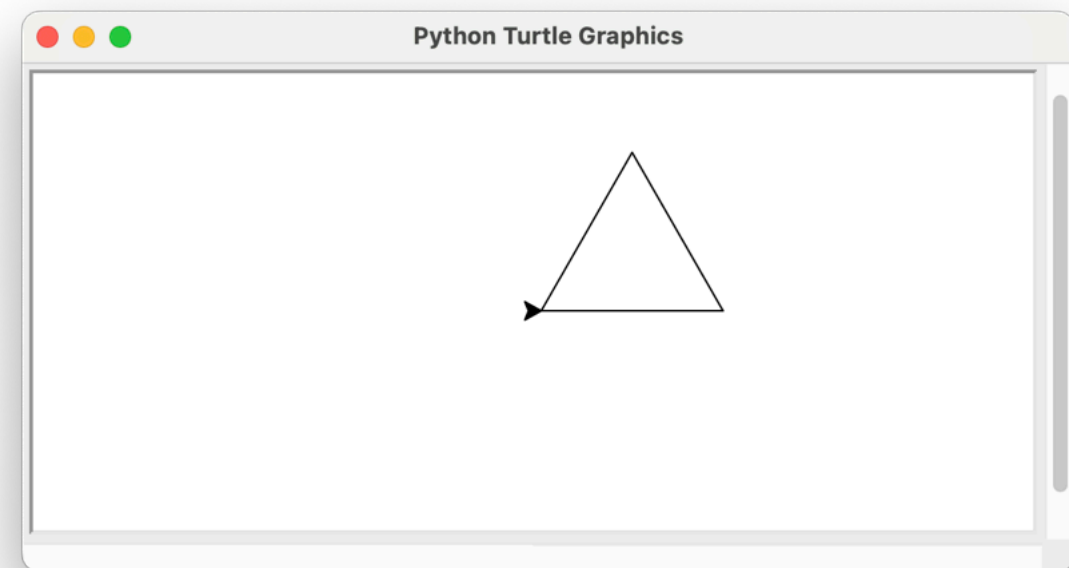


The code for the equilateral triangle is:

```
from turtle import *  
forward(100)  
left(120)  
forward(100)  
left(120)  
forward(100)  
left(120)
```

The forward() and left() commands are repeated 3 times. We can use an iteration:

```
from turtle import *  
for x in range(0,3):  
    forward(100)  
    left(120)
```



Turtle position on the screen

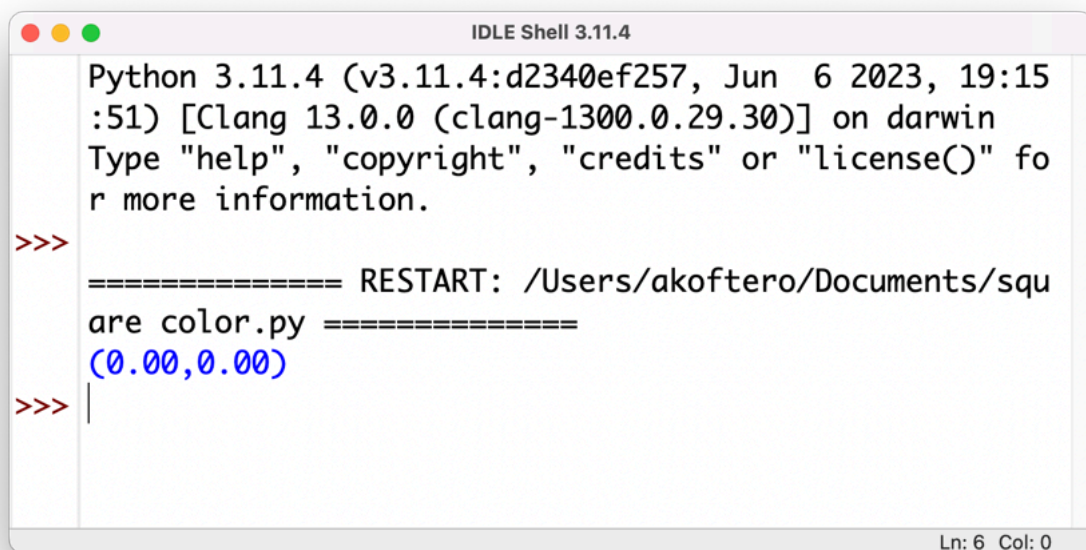
The computer screen consists of rows and columns of pixels, as we saw on the previous page ("Pixels").

Our "turtle" always appears in the center of the screen (of the window to be precise) and "looks" to the right. This position is called "home". It also has the coordinates (0,0).

If we want to see, at any time, the position of the turtle, we use the command below:

```
print(pos())
```

The information will not appear in the graphics display window, but in the Python IDLE window (image below).

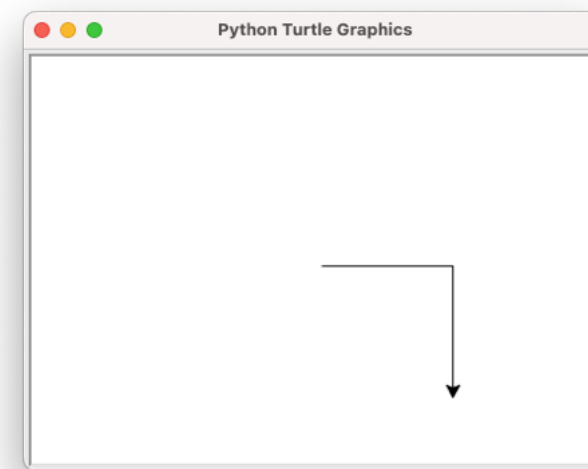


```
Python 3.11.4 (v3.11.4:d2340ef257, Jun 6 2023, 19:15:51) [Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.

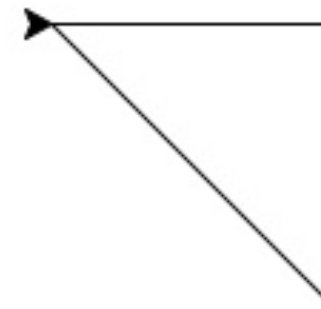
>>>
===== RESTART: /Users/akoftero/Documents/squ
are color.py =====
(0.00,0.00)
>>> |
```

With `home()`, we send our turtle to its original position! Please note! It will not just appear there, it will draw a line all the way to its initial position!

```
forward(100) #draws line right, 100px
right(90) #turn right 90°
forward(100) #draws line down, 100px
```



This can be used cleverly to actually create lines, like the triangle we see below:.

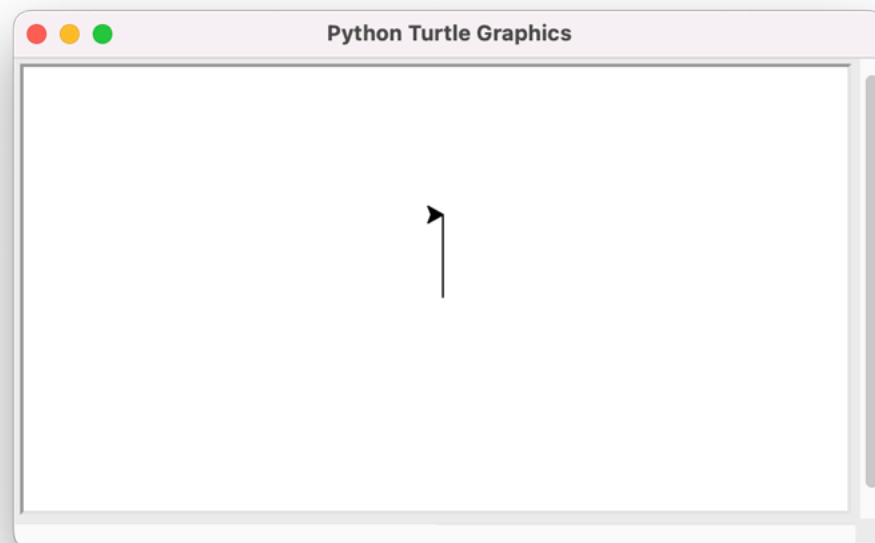


Move on the screen

With the command `setpos()` we can send our turtle to any part of the screen we want. We should remember that `setpos()` needs two pieces of information (numbers): the first number refers to the row we are in, and the second to the column. The starting position is always (0,0).

If we want to "send" our turtle to a different location, then we use the command:

```
setpos(0,50)
```

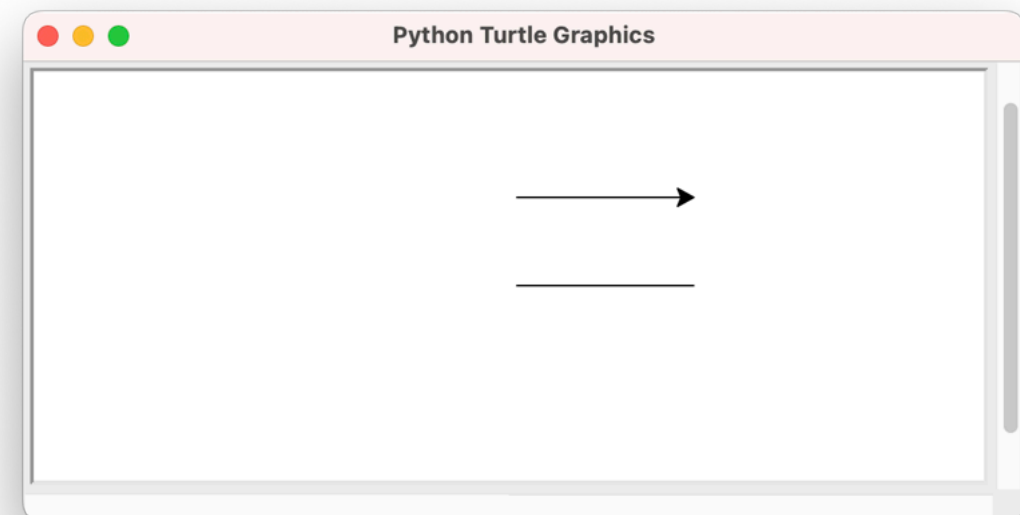


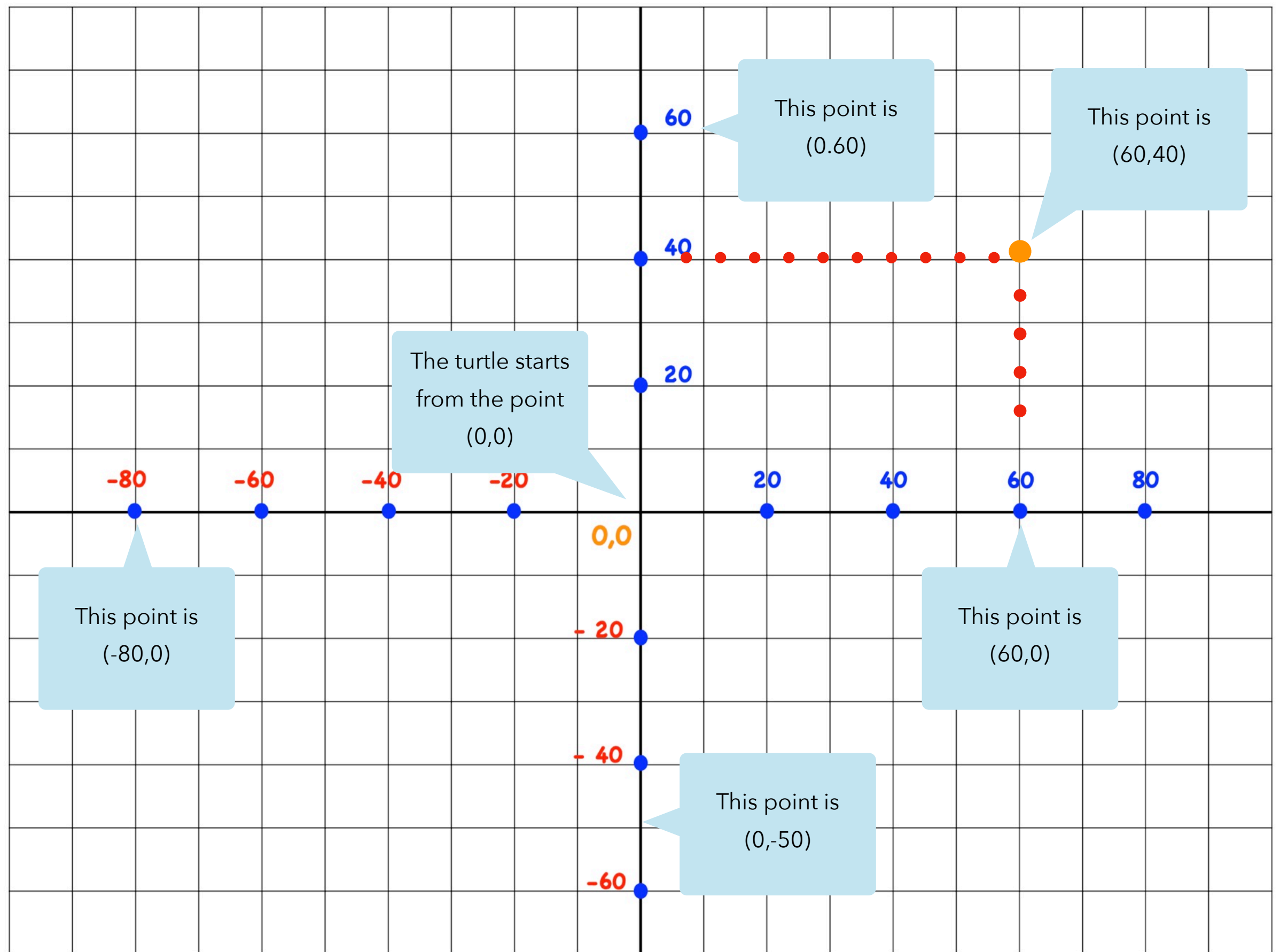
Our turtle moved up 50px, but also drew a line while moving.

If we don't want a line to be created while moving, then we also use the `penup()` and `pendown()` commands.

Let's look at an example:

```
#from the position  
(0,0) it draws line 100px to the right  
forward(100)  
penup() #the pen stops writing  
setpos(0,50) #the pen moves 50px above the original  
position  
pendown() #the pen starts writing  
#from the position (0,50) it draws line 100px to  
the right  
forward(100)
```





Repeating patterns!

On a previous page, we used repetition to create quadrilaterals. In the specific examples, we simplified the commands (reduced the square creation commands by 6).

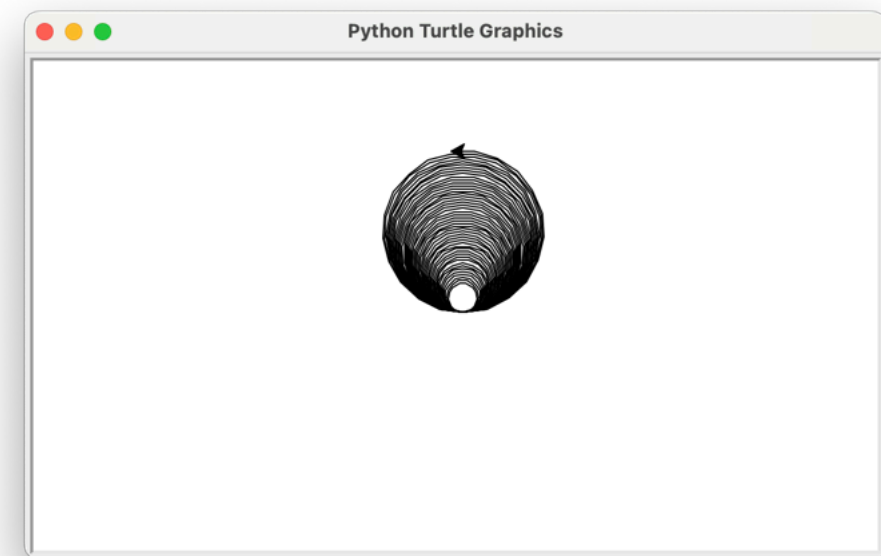
Next we'll use iterations to create a number of shapes that will resize automatically.

```
from turtle import *  
for x in range(10,110):  
    circle(x)
```

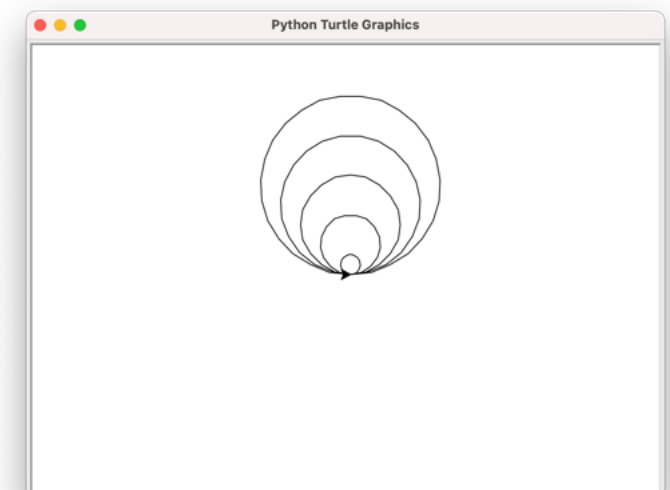


With this repetition, circles will be created with a radius from 10 to 110 pixels, with the use of **range()**.

The repetition continues, with values taken by the x variable (from 10 to 110). But it only increases by 1 radius each time.



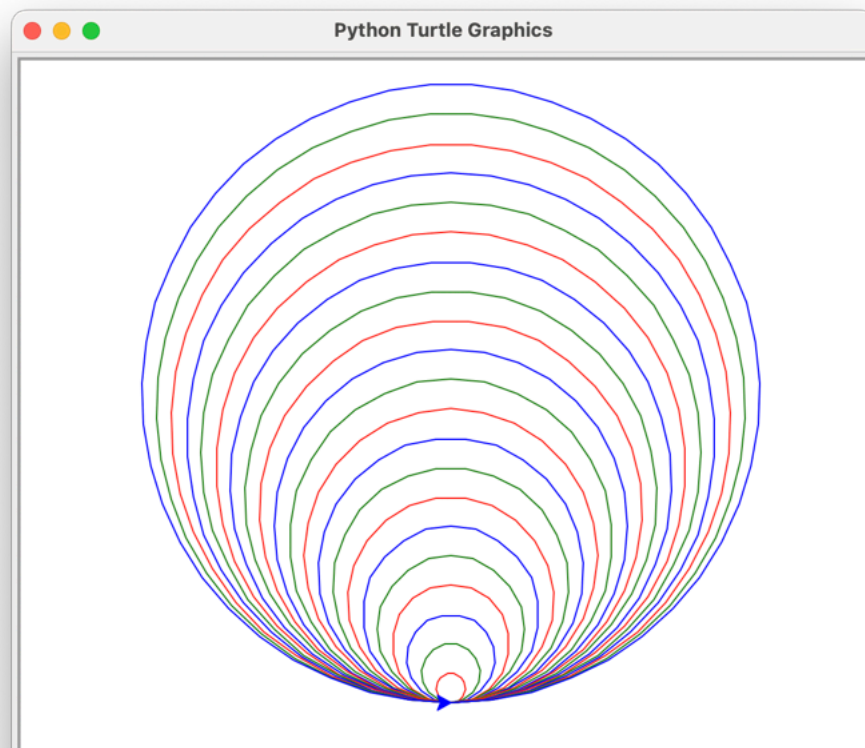
We will give the range() one more **parameter**. The first number (10) indicates which radius to start with. The second number (110) shows where to stop. The third number (20) shows how much to increase the radius each time.



Repetitions & colors

We have created a series of circles, which start from the same point (previous page). But they all have the same color. We will use nested iterations to change the color of each circle each time.

```
from turtle import *
x=10 #initial value of the radius
while x<200:
    for y in ("red","green","blue"):
        circle(x) #forms a circle of radius x
        color(y) #changes color every time
```



```
x=x+10 #radius increases by 10
```

In our example we have repetition within repetition. With `for y in ("red", "green", "blue")`: we repeat the commands it includes 3 times, until all three colors of the parenthesis are used.

The command **circle(x)** will create a circle with radius x (in the second line, we set the initial radius to 10). The **color(y)** command changes the color each time it is repeated (red, then green, then blue).

The command **x=x+10** is necessary so that the radius changes in each repetition, otherwise each circle will be formed on top of the previous one!

The iteration continues until x becomes 200 (`while<200`).

We can experiment with the commands and see how the shape changes. For example, we can change the last line to `x=x+30`. Also, we can add colors (yellow, orange, brown, pink, etc.).



Let's fill with color!

Until now, the shapes we created were colorless inside (or, better, white). We can give color to the inside of a shape, after first - as is logical - creating it.

```
from turtle import *  
circle(50)
```

With the above commands, a circle with a radius of 50 pixels and a white fill color will be created.

To fill a shape, we must first choose the color to use. This can be done with the **fillcolor()** command and the color in parentheses:

```
fillcolor('red')
```

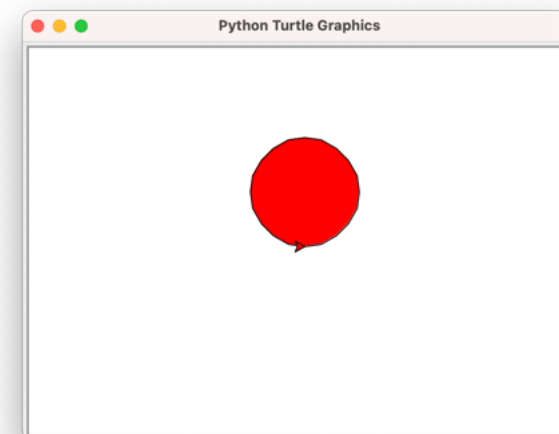


Colors can be selected from a wide range of shades. In this guide we will use the basic colors (red, blue, yellow, green, etc.) and not their shades.



Because it is possible to have several shapes in a program, with different colors, it is very important to "tell" Python, from which shape to start the filling, and at what point to stop.

```
from turtle import *  
fillcolor('red') #we choose a color  
begin_fill() #from this point the filling starts  
circle(50)  
end_fill() #at this point the filling stops
```



Shapes & Functions

Once we understand how the turtle moves on the screen, we can create complex shapes. Let's see how we could create a tree: for its trunk we will draw a rectangle, and for the foliage, a circle.

```
from turtle import *
```

```
fillcolor("brown")#the color is set to brown
```

```
begin_fill() #the filling starts
```

```
for x in range(0,2):
```

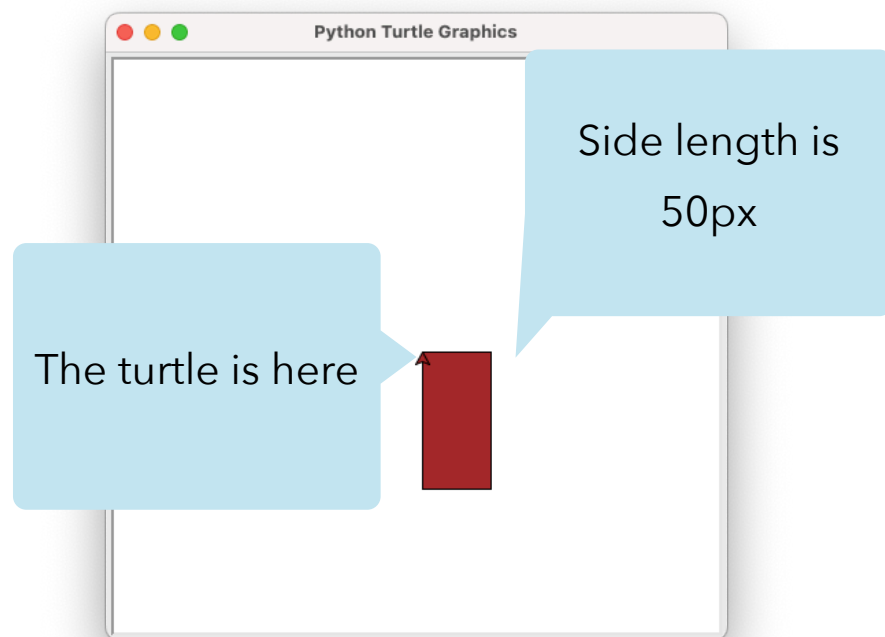
```
    forward(50)
```

```
    right(90)
```

```
    forward(100)
```

```
    right(90)
```

```
end_fill()#the filling is completing
```



The rectangle we created is the trunk of the tree, so we have added brown color to its interior. Next we will also create the upper part of the tree (circle with green fill).

We want the foliage to appear from the middle of the upper side of the rectangle (top right image).

We give the command "**forward**(25)" to go to the middle of the line, and then we create the circle (filled with green color):

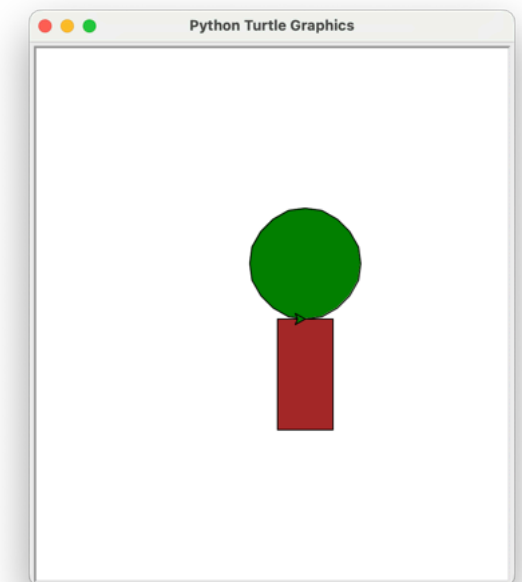
```
forward(25)
```

```
fillcolor("green")
```

```
begin_fill()
```

```
circle(50)
```

```
end_fill()
```



So... how about we use a function to create a row of trees?

All the code we saw on the previous page allows us to create a tree. We will integrate the code into a function, so that we call it every time we want to "plant" a tree.

```
from turtle import *

def mytree()
    #commands to create the trunk
    fillcolor("brown")#brown color is chosen
    begin_fill() #color filling starts
    for x in range(0,2):
        forward(50)
        right(90)
        forward(100)
        right(90)
    end_fill()#color filling stops

    #commands for generating the foliage
    forward(25)
    fillcolor("green")
    begin_fill()
    circle(50) #size of the foliage
    end_fill()
```

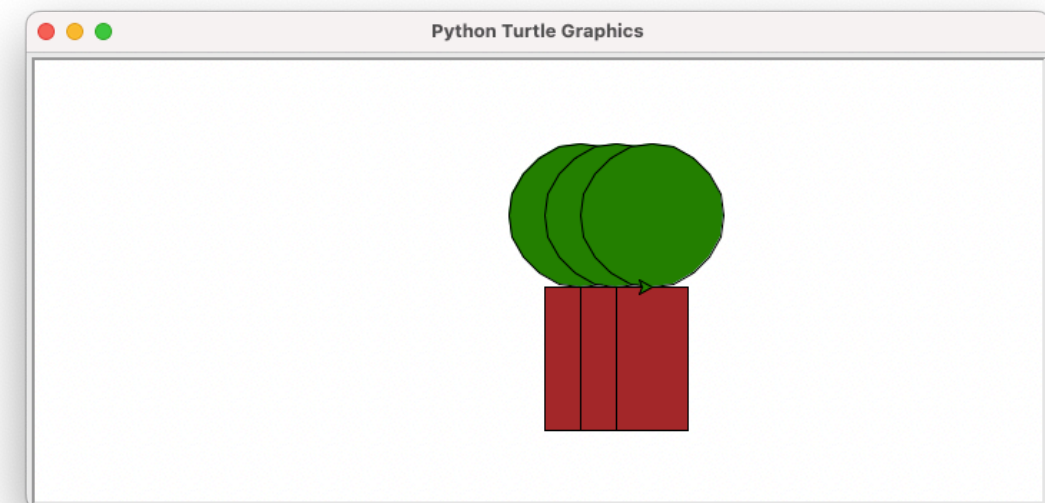
All of the above are the commands that are (now) contained in the **mytree()** function.

Next, we'll "plant" a series of trees on our screen with code.

We add the commands:

```
for x in range(0,3)
    mytree()
```

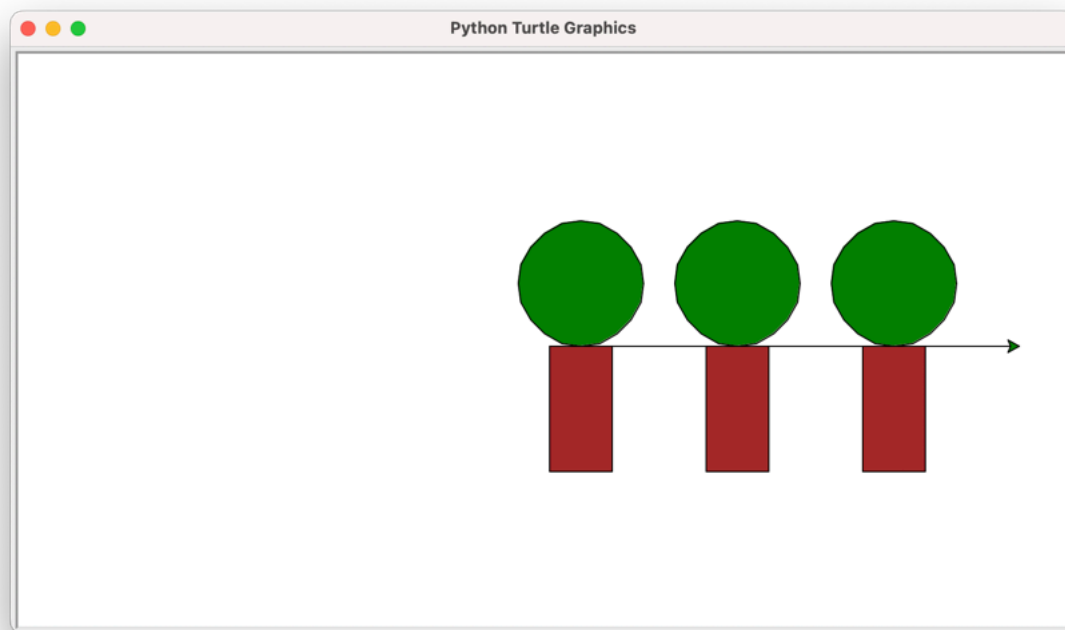
With the above commands, we call the mytree() function 3 times to create 3 trees. If we modify range(0,3) and instead of 3 we put 5, then the function will be repeated 5 times.



The little trees are created, but they are stuck to each other. We need to make one more change to our code so that they appear spaced apart.

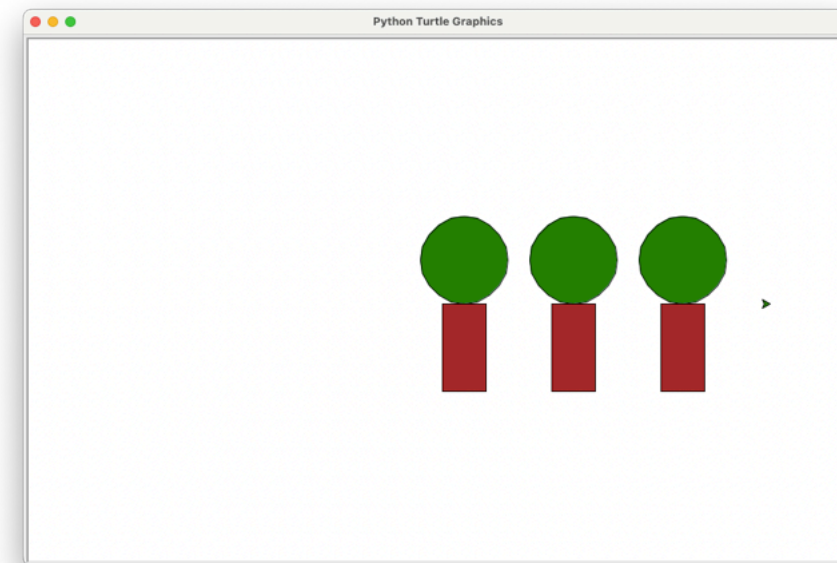
The function we created helped us draw 3 trees in a row (previous page). But these are linked together. We'll use the forward (100) command to keep the distance (you can resize it however you like).

```
for x in range(0,3)
    mytree() #call of the function
    forward(100) #keep the distance 100px
```



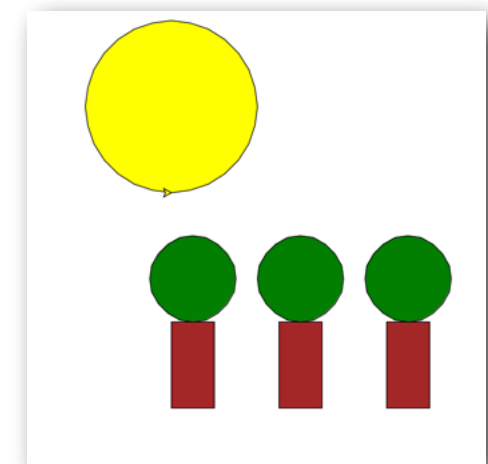
We have kept a distance between the trees, but in moving the turtle he has also drawn a line from one to the other! Easily fixed with the **penup()** and **pendown()** commands.

```
for x in range(0,3)
    mytree() #call of the function
    penup()
    forward(100) #keep the distance 100px
    pendown()
```



Our trees appear in order. If we want to experiment, we can also add a sun.

```
penup()
setpos(0,150)
pendown
fillcolor("yellow")
begin_fill()
circle(50)
end_fill()
```



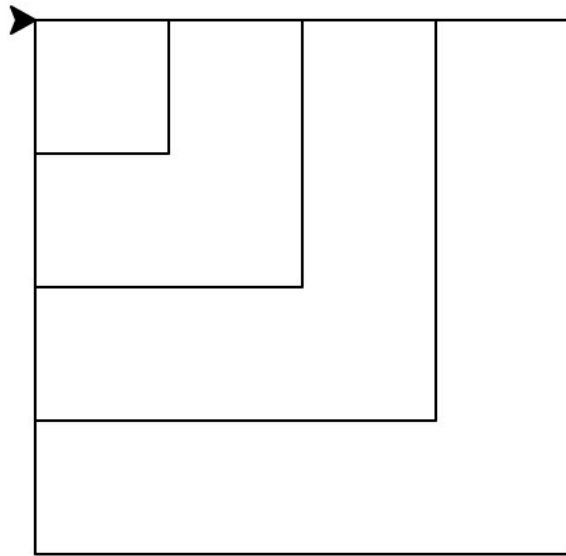
What have we learned so far?

- In **Part C' "Turtles and Pythons"** we worked with additional commands to create geometric shapes.
- We introduced and used the "turtle" feature that allows Python to create geometric shapes.
- We learned how our turtle moves on the screen, starting at (0,0) ("home" position).
- We used movement commands (forward, left, right etc.) to move the turtle and make shapes.
- We leveraged iterations to simplify our code.
- We have created a condition for building complex shapes.



Activities

1. Write a code to create 4 squares of different sizes, one inside the other, as in the figure:



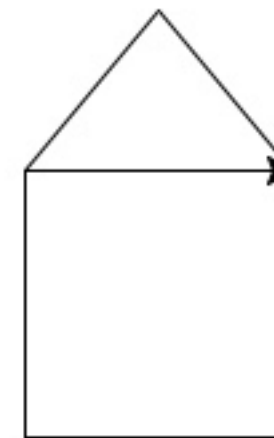
2. Change the code below so that each circle has a different color:

```
from turtle import *
x=10 #initial value of the radius
while x<200:
    for y in ("red","green","blue"):
        circle(x) #initial value of the radius
        color(y) #it changes color every time
        x=x+10 #radius increases by 10
```

3. Draw the shape that will appear when you run the program below:

```
from turtle import *
y=20
for k in range(0,4):
    for x in range(0,4):
        color("blue")
        circle(y)
    y=y+20
```

4. Write the code that creates a house, as shown at the image below:





MINI CV

I was born in Nicosia, Christmas Eve 1974. I got my first computer in 1988, an Atari 520STFM. I started programming almost the same year, with Metacomco BASIC and later with STOS BASIC. I started my studies as an Electrical Engineer at the Higher Institute of Technology, but I started my studies again at the Pedagogical Department of the University of Cyprus. During my studies I acquired my first Macintosh and - with my own loans - Hypercard 2.2 and later Macromedia Director 3.0 studio. I continued my studies with a master's degree in Analytical Programs and Teaching (2007) and a PhD in Information Systems and Communications (2017).

As a teacher, I have been working in schools since 1999, with the exception of the period 2009 - 2011 when I was seconded to the Pedagogical Institute. My interests include online learning environments as well as computer history. With my childhood friend Nikolas Ktenas, we founded in 2014 the first Computer Museum in Cyprus.

Alexandros Kofteros, PhD

alexandros@mathisis.org

<https://www.facebook.com/alexandros.kofteros>

<https://mathisis.org>

PYTHON IN SIMPLE WORDS

Quick tutorial for beginners

Alexandros Kofteros, PhD

This handbook is an introduction to Python for all ages -`range(10,100)`-.

Python is a special programming language. It is a high level language, very easy to learn even by a beginner, but still powerful enough to enable the development of almost any kind of program.

With this book:

- We learn the basic commands of Python
- We work with variables and loops
- We create our own functions
- We draw geometric shapes
- ...we get to love pythons a bit more...

