Algorithms and Software Concepts:
Introduction to programming with python

Q1 AIDAMS Bachelor

Dr. Ahmed El Kerim

Laboratory of Mathematics in Interaction with Computer Science, CentraleSupélec, Paris-Saclay University

Instructor Introduction

- **Instructor:** Dr. Ahmed El Kerim
- **Background:** Research engineer in visualization and computing
- **Contact Information:** Email: ahmed.el-kerim@centralesupelec.fr
- **offices:**Bouygues: sb109 + sc003

Classroom Rules and Guidelines

- **Be on Time**: Please arrive on time for lectures + use sowesign to show your presence.
- **Engage Actively**: Participate actively, don't hesitate to ask questions if you don't understand .
- **Complete Assignments**: Submit assignments and projects by the specified deadlines.
- **Collaborate Responsibly**: Collaborate with others, but respect individual work.
- **Have Fun Learning**: Most importantly, enjoy the learning journey!

Course modalities

Course Structure

- **Duration:** Approximate duration per week : 6 hours
- **Assessments:** Homework (10%), Programming project (30%), and a final exam (60%).

Course Objectives

- Main Objectives:
  - Understand fundamental algorithmic principles.
  - Develop problem-solving skills.
  - Gain in software development concepts.

Course Materials

- **Readings/Resources:** Lecture notes, online tutorials, and coding environments.

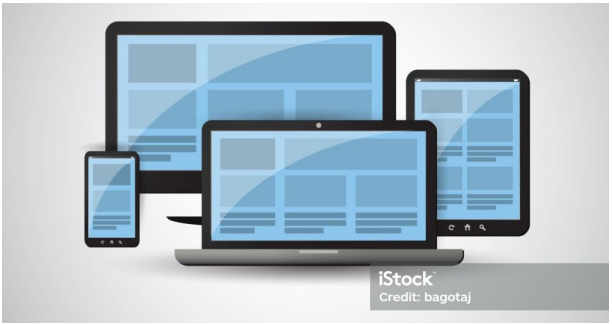
Outline

- Introduction
- Variables and Data Types
- Loop
- Control Structures
- Functions
- Moudule
- File Handling

Table of Contents

- Introduction
  - Computers
  - Algorithms
  - Why Python?
  - Python
- Variables and Data Types
  - Variables
  - Data types
  - Containers
- Loop
  - For Loops
  - While Loops
- Control Structures
- Functions
- Moudule
- File Handling

Understanding the Inner Workings of Computers



- Computers are versatile machines engineered to perform many tasks, from powering our phones and PCs to driving complex systems.
- Computers have profoundly impacted our daily lives, enhancing our efficiency and productivity.
- These remarkable devices execute commands provided by their users.
- Users create programs or software to communicate their instructions to the computer.
- Each computer program or software is meticulously crafted for a specific purpose.
- These software applications are constructed using various programming languages, such as Python.

The Historical Journey of Computers

- 1950s: Enormous computing devices were available to a privileged few.
- 1960s: Large corporations were the primary owners of computers.
- 1970s: Computers started to become smaller and more affordable.
- 1990s: Computers became cheaper and faster, finding their way into most households.

Essential Components of a Computer System

Hardware

Physical parts of the computer, including processor and memory chips, input/output devices, tapes, disks, modems, cable, etc.

- The Processor is used to process data. It is also called Central Processing Unit (CPU). It is the brain of the computer. It consists of electronics circuit. CPU interprets and executes program instructions. All computers must have a Central Processing Unit.
- Storage usually refers to the Secondary storage. The main memory stores data and programs temporary & is called Primary Storage (RAM). The Secondary storage is required to store Data , Information and Programs permanently (main memory).
- A computer system requires different components to perform the functions of Input, Processing, Output and Storage and these four components are necessary for a computer to work.

Essential Components of a Computer System

Sotware

Programs that tell the computer what to do. It provides instructions that the CPU will need to carry out to perform specific tasks categorized into two types:

- System Programs: These control the computer's fundamental operations: Operating system, utilities , user intreface ... etc.
- Application Programs: Written in programming languages, these programs perform specific tasks under the supervision of the operating system: Word processors, games ... etc.

What is Machine Language?

- Machine language is a low level programming language. Machine Language is a combination of one's and zero's (1's and 0's). Machine language/code also called as the Binary language as it only have binary numbers i.e 0's and 1's.
- The computers only understands the 0's and 1's (Binary Values)
- All the programs which are written in high level language programming or Assembly language need to be converted into the Binary code, So that the computer can understand and execute the program.
- Machine Code is bunch of 0's and 1's. So It is almost impossible to do programming in machine language. It is very difficult to read and understand.
- The Machine Code is also the Hardware Dependent. Means if we write any program on machine language which can't be run on other hardware platforms. So Machine code is not portable.

Binary

- Computers are digital, electronic machines that uses binary numbering (base 2) system to transfer, store and manipulate data.
- Binary numbers – Strings of 0's and 1's, are often associated with computers.
- **Question**: Why is this and why can't computers just use base 10 instead of converting to and from binary?
- **Answer**: Modern-day "digital" computer, operates on the principle of two possible states: "on" and "off". The electronic circuit are of two states or bistable using binary logic. Either 5 volts or 0 volts and this is usually represented by binary digits '1' (electrical current present) and '0' (electrical current absent). The "on" state is assigned the value "1", while the "off" state is assigned the value "0".
- Increasing the base will decrease the number of digits required to represent any given number, but today, it is impossible to create a digital circuit that operates in any base other than 2, since there is no state between "on" and "off" (unless you get into quantum computers...).

The number system

Definition

A number system is a way to represent numbers.

- **The denary:** (decimal) number system where the base is ten, meaning we have combination of numbers between 0 to 9 0,1,2,3,4,5,6,7,8,9.
- **The binary:** number system where the base is two, meaning we have only two numbers 0 and 1.

The denary system

Definition

Denary or decimal system is based on the number 10. Some typical denary numbers would be: 58, 96, 80, 3956 etc.

100000	10000	1000	100	10	1
10^5	10^4	10^3	10^2	10^1	10^0

The binary system

Definition

- It is based on the number 2. Some typical binary number would be: 11010110, 011011, 1111000.
- Bit means binary digits '0' and '1'.
- One Nibble is equal to 4 bits. Examples: 0011, 1001, 1111, 1010
- A Byte is a group of 8 bits (2 nibbles), it is used to represent a character. Examples: 1110 0011, 1001 1111, 10101100.

256	128	64	32	16	8	4	2	1
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Converting denary to binary

Example 1 : Convert 25 to binary

- **Step 1:** $32(2^5) > 25 > 16(2^4) \implies 25 - 16 = 9$

16	8	4	2	1
2^4	2^3	2^2	2^1	2^0
1	0	0	0	0

- **Step 2:** $9 - 8(2^3) = 1$

16	8	4	2	1
2^4	2^3	2^2	2^1	2^0
1	1	0	0	0

- **Step 3:** $1 < 4(2^2)$

16	8	4	2	1
2^4	2^3	2^2	2^1	2^0
1	1	0	0	0

- **Step 4:** $1 < 2(2^1)$

16	8	4	2	1
2^4	2^3	2^2	2^1	2^0
1	1	0	0	0

Converting denary to binary

Example 1 : Convert 25 to binary

- **Step 5:** $1 - 1(1^0) = 0$

16	8	4	2	1
2^4	2^3	2^2	2^1	2^0
1	1	0	0	1

- Therefore 25 base 10 (denary) to binary = 00011001
- Check the result by multiplying the 1s in row 3 to corresponding value in row 1. $16 + 8 + 1 = 25$

Exercise

- Convert 60 from denary to binary.
- Convert 246 from dinary to binary.
- Convert 00110110 from binary to dinary.
- Convert 11001101 from binary to dinary.

Results

- $60 \implies 00111100$.
- $246 \implies 11110110$.

The Evolution of Programming Languages

- In the early days, computers were programmed in machine language.
- Assembly languages were developed to simplify programming tasks, introducing mnemonic instructions for easier comprehension.
- An assembler translates assembly language instructions into machine language.
- High-level programming languages have further simplified programming, making it more accessible and closer to spoken languages.
- Notable examples include Python, FORTRAN, C/C++, and Java.
- Compilers translate high-level language code into equivalent machine language.

Low Level Programming Languages | Assembly level language

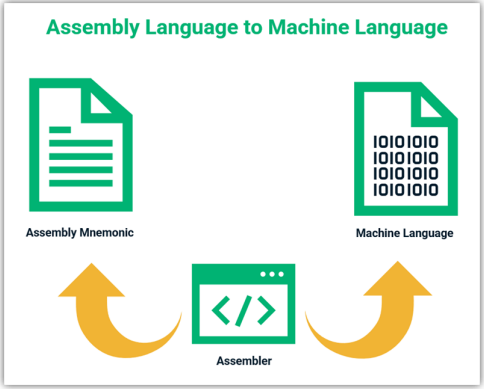
- There are different types of programming languages like low-level and high-level programming languages.
- Programming languages which are in the symbolic format are called as Low-level programming languages. The Symbolic format means usage of hardware specific mnemonics.
- Languages like Assembly language uses the Mnemonics like ADD, SUB, MOV, jmp, etc. These languages are called Low Level Languages. And low level languages are very hard to learn and understand.
- Low level Languages also hardware dependent. The code you wrote for one hardware chipset won't work for other hardware. So you have to re-write your application again for new hardware. The low level languages are not portable.
- Most of the modern programming Languages like Python. Which looks very close to the English language are called High level languages. We can easily learn and master these high level languages and these languages are portable as well.

Key Characteristics of Low Level Programming Languages

- Low level programming languages are in the form of mnemonics like sub, add, mov, jmp, retq, popq, and etc.
- Maintaining code, writing, debugging the low-level code is not easy compared with High level languages code.
- Low-level languages are Hardware dependent languages we cannot reuse code write for one hardware platform on other hardware platform. So these languages are not portable.
- Assembly language is the best example for low level programming language

Translating Assembly language in to Machine language

- The low level language code is close to hardware mnemonics but still it need to convert into the Machine Level Code or Binary code so that computer processor (CPU) will understand it. So We need to translate the assembly code to Machine code which is done by Assembler.



Example Program in Low Level Language:

```c
#include<stdio.h>
int main()
{
    // Take two input numbers
    int a = 10, b = 20;

    // Add two numbers and store result into variable 'sum'
    int sum = a + b;

    // Display the result
    printf("Sum of %d and %d is %d \n", a, b, sum);

    return 0;

}
```

C Program to Add two numbers: the sum of two variables 'a'and 'b'.

```
.section __TEXT,__text,regular,pure_instructions
.build_version macos, 10, 15 sdk_version 10, 15, 6
.globl _main                    ## -- Begin function main
.p2align 4, 0x90
_main:                                  ## @main
.cfi_startproc
## %bb.0:
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq %rsp, %rbp
.cfi_def_cfa_register %rbp
subq $32, %rsp
movl $0, -4(%rbp)
movl $10, -8(%rbp)
movl $20, -12(%rbp)
movl -8(%rbp), %eax
addl -12(%rbp), %eax
movl %eax, -16(%rbp)
movl -8(%rbp), %esi
movl -12(%rbp), %edx
movl -16(%rbp), %ecx
leaq L_.str(%rip), %rdi
movb $0, %al
callq _printf
xorl %ecx, %ecx
movl %eax, -20(%rbp)        ## 4-byte Spill
movl %ecx, %eax
addq $32, %rsp
popq %rbp
retq
.cfi_endproc
                                        ## -- End function
.section __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
.asciz "Sum of %d and %d is %d \n"


.subsections_via_symbols
```

The corresponding assembly language code, which contains lot of mnemonics which are hard to remember.

High Level Programming Languages

- The Programming Languages which are syntactically similar to english and easy to understand and Independent of Hardware are referred as High level programming languages.
- High level languages are easy to learn and understand. These languages are close to developer compared to the hardware. Due to this we can rapidly design and develop new applications.
- But We need to translate the code written in the high level language to Machine code before running on the hardware. This process is called Compelling or Interpreting depending upon the language you are using.
- The High level Languages are relatively slow compared to the low level languages due to compilation overhead.

Translation of High level programming language into machine level language

- Any High level programming language need to be translated into the Binary Language, So that the Computer processor can understand and run the program.

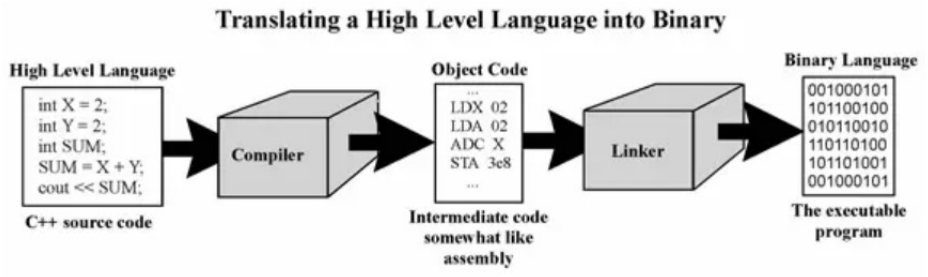


Image Credits : uah.edu

Translation of High level programming language into machine level language

- First of all, We need to create a sample program in High level language such as c++, Java, etc.
- As the program is written in High level language, The computer processor won't understand high level language, We need to convert it to machine code.
- The first step is the Compiler. The compiler will compile the code. Which converts high level language into the low level code(Object file). This object file is Intermediate code which will be in the low level language such as Assembly language.
- Then Linker will link all the external libraries and files you have used in your program and link them together to create an Executable file.
- The final Executable file will be in the Binary Language or Machine Language. This executable file will be run by the processor.

Remark

All these steps are done automatically once you trigger the compilation. and Compiler will perform all above steps behind the scenes and generates the executable file.

The Problem-Solving Process in Computing

- The process of solving problems with computers involves several key steps:
- 1. State the Problem
- 2. Analyze the problem: Outline solution requirements and design an algorithm.
- 3. Design an algorithm to solve the problem.
- 4. Implement the algorithm in a programming language like Python and verify its functionality.
- 5. Maintain the program: Use and modify it if the problem domain evolves.

Algorithms

Definition

- An algorithm is any set of instructions that specifies a series of steps to solve a problem correctly.
- Multiple algorithms may exist to solve a given problem with varying levels of efficiency.

Algorithms and Programs

- An algorithm is a finite set of instructions that explains a step-by-step solution to a problem.
- Computers can execute algorithms by following a finite number of steps or instructions.
- A program is a collection of computer instructions that implements an algorithm.

Why Should I Write Algorithms?

- Writing algorithms is essential when developing computer programs for scientific or problem-solving tasks.
- Understanding the steps to solve a problem is a fundamental prerequisite for writing a computer program.

Why Should I Write Algorithms?

- To write a computer program effectively, you must clearly understand the algorithm that outlines the required steps to solve your problem.
- The sequence and order of these steps are critical to writing a correct algorithm.

Importance of Precision

- The sequence or order of steps in an algorithm is crucial.
- Precision is essential when specifying an algorithm to be translated into a computer program.
- Minor differences in expressions like A*B+C and (A*B)+C can have significant implications, and computers follow instructions precisely.

Problem Solving Methodology and Algorithms

- Problem Specification:
  - Clearly define the problem.
- Analysis:
  - Determine input and output requirements.
  - Analyze how to transform input into output.
- Design:
  - Develop a step-by-step method (algorithm).
- Test Plan:
  - Define tests to verify the algorithm's correctness.
- Implementation:
  - Code the algorithm into a program.
- Testing:
  - Execute tests to ensure the program works as intended.
- Refinement:
  - Make improvements or optimizations if necessary.

Problem Specification I

- Any problem-solving process involves:
  - Input → Algorithm → Output
- Determine what information is available as input to your algorithm.
- Determine what information is desired as output from your algorithm.

Specification and Analysis

- Determine the steps needed to transform the input data into the desired output.
- Enumerate all special cases that the algorithm must handle.
- Modify or redesign the algorithm to handle all special cases.

Verifying Algorithms

- Before translating an algorithm into a program, verify that it produces the desired results.
- Ensure that all special cases are addressed.
- Confirm that the algorithm terminates after determining the outputs.

Summary: Writing Algorithms

- To write successful algorithms:
  - Begin by understanding the problem, input data, and desired output.
  - Determine a series of steps to transform input data into output results.
  - Enumerate and address all special cases.
  - Verify the algorithm's correctness.

Example: Problem Specification

- Scenario: Preparing coffee for friends during a weekend getaway.
- Your task: Make coffee.
- Initial information: Coffee is in the freezer.

Example: Analysis and Design

- Observation: Coffee makers and coffee filters are available.
- Subdividing the problem:
  - Take the coffee out of the freezer.
  - Put coffee in a filter.
  - Put the filter in the coffee maker.
  - Put water in the coffee maker.
  - Turn on the coffee maker.
  - Put the rest of the coffee back in the freezer.

Example: Algorithm Refinement I

- Discovery: Coffee beans found in the freezer instead of ground coffee.
- Refinement of step 2:
  1. Find the coffee grinder.
  2. Put the coffee beans into the grinder.
  3. Grind the coffee beans.
     - Stop grinding.
     - Check if the coffee beans are properly ground.
     - Continue grinding if needed.
     - Repeat until the coffee is properly ground.
  4. Put the ground coffee in the filter.

Example: Algorithm Refinement II

- Additional consideration: Determining when the coffee is properly ground.
- Further refinement of step 2c (Grind the coffee beans):
  - Stop grinding.
  - Check to see if the coffee beans are properly ground.
  - Continue grinding if they are not.
  - Repeat until the coffee is properly ground.

Example: Refined Algorithm I

- Take the coffee out of the freezer.
- Put coffee in a filter:
  - Find the coffee grinder.
  - Put the coffee beans into the grinder.
  - Grind the coffee beans:
    - Stop grinding.
    - Check to see if the coffee beans are properly ground.
    - Continue grinding if needed.
    - Repeat until the coffee is properly ground.
  - Put the ground coffee in the filter.
- Put the filter in the coffee maker.
- Put water in the coffee maker.
- Turn on the coffee maker.
- Put the rest of the coffee back in the freezer.

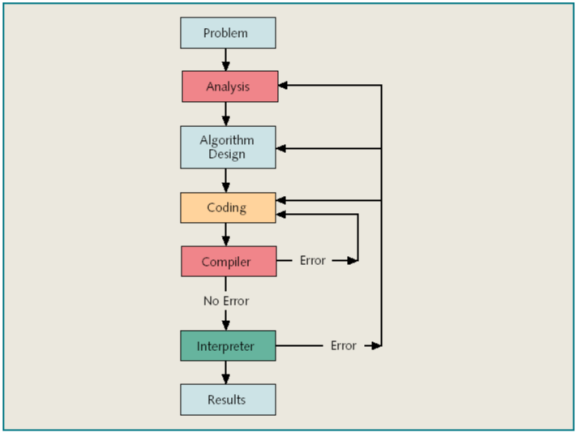
Choices in Algorithm Design

- Multiple algorithms may exist to solve a given problem.
- How do we determine which algorithm is the best choice?

Class Discussion

- In five minutes, set up an algorithm for solving a specific problem.

Problem-Analysis-Coding-Execution Cycle



Programming Methodologies

- Two basic approaches to programming design:
  - Structured Design:
    - Divide a problem into smaller subproblems.
    - Solve each subproblem.
    - Combine the solutions to solve the main problem.
  - Object-Oriented Design (OOD):
    - Program is a collection of interacting objects.
    - Objects consist of data and operations.
    - Steps in OOD:

What is Python?

- Python is a versatile t is very versatile, thanks to numerous modules covering various domains: Numpy for linear algebra, matrices, vectors, linear systems...Scipy for probability/statistics, 1D and 2D FFT, digital filtering, images... Matplotlib for plotting, OS for directory and file manipulation... etc.
- It's often referred to as both a programming (database access, object-oriented programming), a scripting language (file manipulation, system administration, machine configuration) and in scientific computing (mathematical libraries).
- Python is classified as an interpreted language executed line by line.
- Python is a recent language (1989) widely used in web programming as a scripting language ,

Differences between Program and Scripting Languages

- Program:
  - A program is executed after source code is compiled into an executable.
  - Programs consist of a sequence of instructions for a computer to perform specific tasks.
- Scripting:
  - A script is interpreted and executed line by line.
  - Scripts are written in scripting languages and control other software applications.

()

Interpreted programming language

- Python is an interpreted programming language, meaning that the instructions you send to it are "translated" into machine language as they are read. Other languages, such as C/C++, are "compiled languages" because, before they can be executed, specialized software transforms the program code into machine language. This step is called "compilation." With each code modification, a compilation step needs to be invoked.
- It saves a considerable amount of time during program development because no compilation or linking is required. But a compiled language tends to be faster than an interpreted one (the translation of your program slows down execution), although this difference tends to decrease over time with improvements.

History of Python

- Guido Van Rossum developed Python in the late 1980s at the National Research Institute for Mathematics and Computer Science in the Netherlands.
- It was first published in 1991.
- The name "Python" came from the TV show "Monty Python's Flying Circus.
- Python has always been open source, fostering a vibrant community of contributors.

Why Was Python Created?

"My original motivation for creating Python was the perceived need for a higher-level language in the Amoeba [Operating Systems] project. I realized that the development of system administration utilities in C was taking too long. Moreover, doing these things in the Bourne shell would only work for some reasons. So, there was a need for a language that would bridge the gap between C and the shell."

- Guido Van Rossum

Key Features

- Simplicity:
  - Python's syntax is designed to be simple and readable, resembling English sentences.
  - This simplicity enhances code clarity and reduces the cognitive load on programmers.
- Ease of Learning:
  - Python uses a small number of keywords and has a straightforward program structure.
  - Programmers familiar with languages like C find it easy to migrate to Python.
- Open Source:
  - Python is an open-source language freely available for download and use.
  - It encourages collaboration and has a vast library of open-source packages.
- High-level Language:
  - Python is considered a high-level language because its syntax is closer to human language than low-level languages like assembly.
- Dynamically Typed:
  - In Python, you don't need to declare variable types; they are determined dynamically at runtime.
  - This flexibility allows a variable to change its type during program execution.

Table of Contents

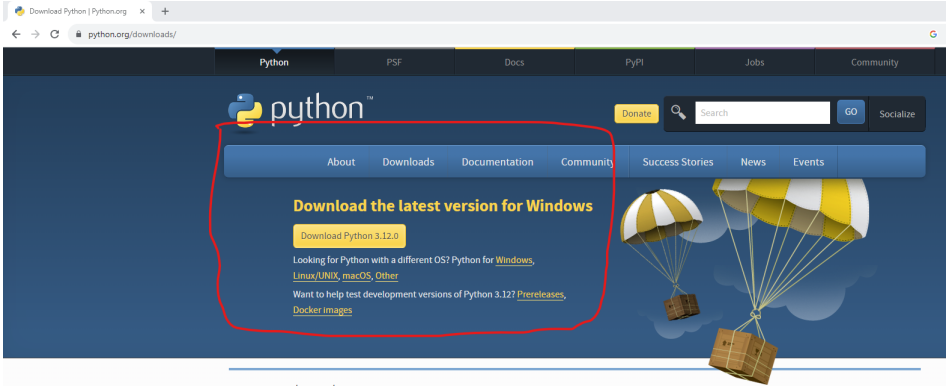
- Introduction
  - Computers
  - Algorithms
  - Why Python?
  - Python
- Variables and Data Types
  - Variables
  - Data types
  - Containers
- Loop
  - For Loops
  - While Loops
- Control Structures
- Functions
- Moudule
- File Handling

Python Programming Language

- Python programming language was created in 1989 by Guido van Rossum in the Netherlands.
- The name "Python" is a tribute to the Monty Python's Flying Circus TV show, of which G. van Rossum is a fan.
- The first public release of Python was in 1991.
- The latest version of Python is version 3.
- Python version 2 is obsolete and no longer maintained; avoid using it.
- The Python Software Foundation is responsible for Python's development and community engagement.
- Python has several interesting features:
  - It is cross-platform, running on various operating systems, from Windows and Mac OS X to Linux.
  - It is free and can be installed on as many computers as needed.
  - It is a high-level language, requiring relatively little knowledge of computer internals.
  - It is interpreted, meaning Python scripts do not need to be compiled before execution, unlike languages such as C or C++.
  - It is object-oriented, allowing the design of entities that mimic real-world objects with specific rules and interactions.
  - It is relatively easy to learn.
  - It is widely used in data science and scientific computing.
- Python is now taught in many programs, from high school to higher education.

Installing Python

- Python is pre-installed on most Unix systems, including Linux and macOS, making it readily available for development.
- For Windows users, you can download the latest version from `https://www.python.org/downloads/`.



- The Python website provides installation instructions and packages for various platforms.

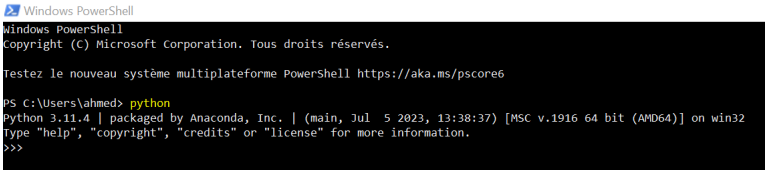
Launching Python: Interactive Shell and Scripts

Interactive Shell

- A shell is an interactive command-line interpreter facilitating computer interaction.
- Use the shell to launch the Python interpreter.
- Commands:
  - **Windows:** Just run `python`
  - **Unix:** Run `python` or `python3`

Exploring Python: Interactive Interpreter

- Python is an interpreted language, meaning each line of code is read and interpreted before being executed by the computer.
- Open a shell and execute the command: `python`
- This launches the Python interpreter, and you should see something like:



- The triple chevron (>>>) is the Python interpreter prompt, where you can enter commands.

```
PS C:\Users\ahmed> python
Python 3.11.4 | packaged by Anaconda, Inc. | (main, Jul  5 2023, 13:38:37) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> r = 10
>>> A = 3.14*r*r
>>> print(A)
314.0
>>>
```

- To exit the interpreter: Type `exit()` and press Enter.

Beyond the Interpreter: Python Scripts

- While the interpreter is useful, it has limitations, especially when handling more sequences of instructions.
- Similar to other programming languages, we can store these instructions in a file, commonly called a Python script or program.
- To illustrate, open a text editor, write your code
- Save the file with the name `test.py` and close the text editor.
- **Note:** Python scripts typically use the standard file extension `.py`.
- To run your script, open a shell and execute the command: **python test.py**.

Remark

- In a script, anything following the character # is ignored by Python until the end of the line and is considered a comment.
- Comments should provide explanations of your code in human-readable language.
- Here's an example:

```
1 # Your first comment in Python.
2 print("Hello world!")
3
4 # Other more useful commands could follow.
```

Choosing a Text Editor for Python

- Learning a computer language like Python involves writing lines of code using a text editor.
- For beginners, recommended text editors include:
  - Notepad++ on Windows
  - BBEdit or CotEditor on Mac OS X
  - Gedit on Linux
- If you prefer other editors such as Atom, Visual Studio Code, Emacs, Vim, Pycharm, etc., feel free to use them!
- It's important to note that software like Microsoft Word, WordPad, and LibreOffice Writer are not text editors; they are word processors and should not be used for writing code.

Table of Contents

- Introduction
  - Computers
  - Algorithms
  - Why Python?
  - Python
- Variables and Data Types
  - Variables
  - Data types
  - Containers
- Loop
  - For Loops
  - While Loops
- Control Structures
- Functions
- Moudule
- File Handling

Variables

Definition

- A variable is a portion of the computer's memory where a value is stored. From the programmer's perspective, this variable is defined by a name, while for the computer, it is essentially an address — a specific region in memory.
- In Python, the declaration of a variable and its initialization (i.e., the initial value to be stored in it) coincide.

Variables

- Example:

```
1 >>> x = 2
2 >>> x
3 2
```

- Line 1: In this example, we declared and initialized the variable `x` with the value 2.
- What is happening behind the scence?
  - Python "guessed" that the variable was an integer. Python is dynamically typed.
  - Python allocated (reserved) memory space to hold an integer. Each variable type occupies varying amounts of memory. Python also ensured we could reference the variable using the name `x`.
  - Finally, Python assigned the value 2 to the variable `x`.
- Lines 2 and 3: The interpreter allowed us to see the content of the variable by simply typing its name.
- The assignment operator = works in a specific way: `x = 2` means assigning the value on the right side of the = operator (in this case, 2) to the variable on the left side (here, `x`).
- The statement `x = y - 3`, the operation `y - 3` is evaluated first, and then the result is assigned to the variable `x`.

Variables Data Types

- The type of a variable corresponds to its nature. The main types are:
  - **Int**: Integers - represent whole number values.
  - **Float**: Float data type represents decimal point values.
  - **Complex**: Complex numbers. They are built from floats or int: complex(re, im). One can also use the pure imaginary number 1j.
  - **Boolean**: the result is either true or false.
  - **Strings**: character (string or str)
- In the previous example, we stored an integer (int) in the variable x, but it's entirely possible to store floats, character strings (string or str), complex, etc.
- Example:

```
1 >>> y = 3.14
2 >>> y
3 3.14
4 >>> a = "Hello"
5 >>> a
6 'Hello'
```

General remarks

- Python automatically recognizes certain variable types (integer, float). However, a character string must be enclosed in quotes (double or single) to indicate the string's beginning and end to Python.
- In Python, as in most programming languages, the point "." is used as the decimal separator. For example, 3.14 is recognized as a float in Python, whereas this is not the case for 3,14.
- Variable names in Python can consist of lowercase letters (a to z), uppercase letters (A to Z), numbers (0 to 9), or the underscore character (_). Spaces are not allowed in variable names.
- A variable name should not start with a digit, and it is not recommended to start it with the underscore character (_).
- It is crucial to avoid using a "reserved" word in Python as a variable name (e.g., print, range, for, from, etc.).
- Python is case-sensitive, meaning variables such as TesT, test, and TEST are considered distinct.
- We can represent large or small numbers using powers of 10 with the symbol e:

```
1 >>> 1e6
2 1000000.0
3 >>> 3.12e-3
4 0.00312
```

- Writing 1e6 or 3.12e-3 does not imply the use of the mathematical constant e, but signifies 1×10^{6} and 3.12×10^{-3} respectively.

Arithmetic Operations

- The four basic arithmetic operations are straightforward for integers, floats and complexs:

```
1 >>> x = 45
2 >>> x + 2
3  47
4 >>> y = 2.5
5 >>> x - y
6  44.5
7 >>> a = 1 + 1j
8 >>> b = 2 + 2j
9 >>> c = a * b
10 >>> c
11  4j
12 >>> 3 / 4
13   0.75
```

Arithmetic Operations

- The power operator uses the symbol **:

```
1 >>> 2**3
2 8
3 >>> 2**4
4 16
5 >>> pow(3, 3)
6 >>> 9
```

- To obtain the quotient and the rest of an integer division, the symbols // and % are used, respectively:

```
1 >>> 5 // 4
2 1
3 >>> 5 % 4
4 1
5 >>> 8 // 4
6 2
7 >>> 8 % 4
8 0
```

Arithmetic Operations

- Finally, there are "combined" operators that perform an operation and assignment in a single step:

```
1 >>> i = 0
2 >>> i = i + 1
3 >>> i
4 1
5 >>> i += 1
6 >>> i
7 2
8 >>> i += 2
9 >>> i
10 4
```

- The += operator performs addition and assigns the result to the same variable. This operation is called an "increment." The -=, *=, and /= operators behave similarly for subtraction, multiplication, and division.

String operations

- For strings, two operations are possible: addition and multiplication:

```
1 >>> c = " Hello"
2 >>> c
3  'Hello'
4 >>> Hello + "Python"
5  'Hello Python'
6 >>> c * 3
7  'HelloHelloHello'
```

- The addition operator "+" concatenates (joins) two strings.
- The multiplication operator "*" between an integer and a string duplicates (repeats) a string several times.

Booleans operations

- Boolean operations return 0 or False, or 1 or True.

Operations	Interpretation
$<$	Strictly less than
$<=$	Less than or equal to
$>$	Strictly greater than
$>=$	Greater than or equal to
$==$	Equal
$!=$	Not equal to
is	Object identity
is not	Object non-identity.

Checking Variable Types

- If you forget the type of a variable, you can use the `type()` function to remind you:

```
1 >>> x = 2
2 >>> \texttt{type(x)}
3 <class 'int'>
4 >>> y = 2.0
5 >>> \texttt{type(y)}
6 <class 'float'>
7 >>> z = '2'
8 >>> \texttt{type(z)}
9 <class 'str'>
```

Type Conversion

- To convert types, i.e., switch from a numeric type to a string or vice versa. In Python, the functions `int()`, `float()`, and `str()` do the conversion. examples:

```
1 >>> i = 3
2 >>> \texttt{str(i)}
3 '3'
4 >>> i = '456'
5 >>> \texttt{int(i)}
6 456
7 >>> \texttt{float(i)}
8 456.0
9 >>> i = '3.1416'
10 >>> \texttt{float(i)}
11 3.1416
```

General remarks

- We encountered several functions, including `type()`, `int()`, `float()`, and `str()`.
- Functions in Python follow a general syntax: `function()`.
- The content within the parentheses of a function is called an argument, and it is what we "pass" to the function as an argument or input.
- For example, in the statement `type(2)`, the integer 2 is the argument passed to the `type()` function.
- A function is like a box to which we provide one (or more) argument(s); it performs an action, and it can return a result. For instance, the `type()` function returns the variable type passed as an argument.

Functions examples

In Python, there are built-in functions `min()` and `max()` that respectively return the minimum and maximum values among multiple integers and floats:

```
>>> min(1, -2, 4)
-2
>>> pi = 3.14
>>> e = 2.71
>>> max(e, pi)
3.14
>>> max(1, 2.4, -6)
2.4
```

Functions examples

- the print() function shows the content of the argument passed between parentheses.
- Basic usage of print() function:

```
1 >>> print ("Hello world!")
2 'Hello world!'
```

- More examples with multiple variables:

```
1 >>> x = 32
2 >>> nom = "John"
3 >>> print ( name , "has", x , " years ")
4 'John has 32 years'
```

- Concatenation and sep examples:

```
1 >>> ani1 = "cat"
2 >>> ani2 = "mouse"
3 >>> print ( ani1 , ani2 )
4 'cat mouse'
5 >>> print ( ani1 + ani2 )
6 'catmouse'
7 >>> print ( ani1 , ani2 , sep ="")
8 'catmouse'
```

Lists

- **Definition**
  - A list is a data structure that contains a series of values.
  - Python allows the construction of lists containing different types of values (e.g., integers and strings), providing flexibility.
  - Declared by values separated by commas and enclosed in square brackets.
- **Examples**
  - `animals = ["giraffe", "tiger", "monkey", "mouse"]`
  - `sizes = [5, 2.5, 1.75, 0.15]`
  - `mixed = ["giraffe", 5, "mouse", 0.15]`
- **Using Lists**
  - Elements in a list can be accessed by their position (index).
  - Indices start from 0 to n-1 for a list with n elements.
  - Example: `animals[0]`, `animals[1]`, `animals[3]`
  - Beware of index errors (e.g., `animals[4]`), indices start at 0!

List Operations

- **Concatenation and Duplication**
  - Lists support the + operator for concatenation and the * operator for duplication.
  - Example:
    - `ani1 = ["giraffe", "tiger"]`
    - `ani2 = ["monkey", "mouse"]`
    - `ani1 + ani2` ⇒ `["giraffe", "tiger", "monkey", "mouse"]`
    - `ani1 * 3` ⇒ `["giraffe", "tiger", "giraffe", "tiger", "giraffe", "tiger"]`
- **Appending Elements**
  - The + operator is useful for concatenating two lists.
  - `.append()` method adds a single element to the end of a list.
  - Example:
    - **Create an empty list**: `a = []`
    - **Concatenation**: `a = a + [15]`, `a = a + [-5]`
    - **Using** `.append()`: `a.append(13)`, `a.append(-3)` :
    - **Result**: `a = [15, -5, 13, -3]`

Negative Indexing in Lists

- Lists can be indexed with negative numbers, following the model:
  - List: `["giraffe", "tiger", "monkey", "mouse"]`
  - Positive indices: 0 1 2 3
  - Negative indices: -4 -3 -2 -1
- Negative indices count from the end of the list.
- Advantages include easy access to the last element without knowing the list's length.
- Example:
  - `["giraffe", "tiger", "monkey", "mouse"]`
  - `animals[-1]` ⇒ `'mouse'`
  - `animals[-2]` ⇒ `'monkey'`

List Slices

- **List Slices**
  - Lists can select a portion of the list using slicing with the pattern `[m:n+1]`.
  - This retrieves all elements from the m-th (inclusive) to the $n+1$-th element (exclusive).
- **Examples of List Slices**
  - `animals = ["giraffe", "tiger", "monkey", "mouse"]`
  - `animals[0:2]` ⇒ `['giraffe', 'tiger']`
  - `animals[0:3]` ⇒ `['giraffe', 'tiger', 'monkey']`
  - `animals[1:]` ⇒ `['tiger', 'monkey', 'mouse']`
  - `animals[:]` ⇒ `['giraffe', 'tiger', 'monkey', 'mouse']`
  - `animals[1:-1]` ⇒ `['tiger', 'monkey']`
- **Default Slicing and Specifying Step**
  - If no indices are specified, Python defaults to all elements from the beginning to the end, respectively.
  - Specify the step by adding an extra colon and indicating the step as an integer.
  - Examples:
    - `x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
    - `x[::1]` ⇒ `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
    - `x[::2]` ⇒ `[0, 2, 4, 6, 8]`
    - `x[::3]` ⇒ `[0, 3, 6, 9]`
    - `x[1:6:3]` ⇒ `[1, 4]`

List functions

- The `len()` function provides the length of a list, i.e., the number of elements in the list.
  - Example:
    - `animals = ["giraffe", "tiger", "monkey", "mouse"]`
    - `len(animals)` ⇒ 4
    - `len([1, 2, 3, 4, 5, 6, 7, 8])` ⇒ 8
- `range()` **Function and** `list()` **Function**
  - The `range()` function generates integers within a specified range.
  - The `range()` function works with the pattern `range([start,] stop[, step])`, and the square brackets denote optional arguments.
  - When used in combination with the `list()` function, it produces a list of integers.
  - Example:
    - `list(range(10))` ⇒ `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
    - `list(range(15, 20))` ⇒ `[15, 16, 17, 18, 19]`
    - `list(range(0, 1000, 200))` ⇒ `[0, 200, 400, 600, 800]`
    - To generate a list of decreasing integers, specify a negative step:
    - `list(range(2, -2, -1))` ⇒ `[2, 1, 0, -1]`

List Functions

- Python provides functions to operate on lists, including `min()`, `max()`, and `sum()`.
- These functions are used to find the minimum, maximum, and sum of the elements in a list.
- **Example: Given a list** `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
  - `sum(liste)` $\Rightarrow$ `45`
  - `min(liste)` $\Rightarrow$ `0`
  - `max(liste)` $\Rightarrow$ `9`

Lists of Lists

- **Introduction**
  - Lists in Python can contain elements of various types, including other lists.
  - Constructing lists of lists can be practical for organizing structured data.
- **Example:**
  - `enclosure1 = ["giraffe", 4]`
  - `enclosure2 = ["tiger", 2]`
  - `enclosure3 = ["monkey", 5]`
  - `zoo = [enclosure1, enclosure2, enclosure3]`
  - `zoo` $\Rightarrow$ `[['giraffe', 4], ['tiger', 2], ['monkey', 5]]`
- **Accessing Elements**
  - To access an element in the list, use regular indexing: `zoo[1]` $\Rightarrow$ `['tiger', 2]`.
  - To access an element in the sub-list, use double indexing:
  - `zoo[1][1]` $\Rightarrow$ 2

Dictionaries

Definition

Dictionaries help handle complex structures, providing a way to map keys to values. They are unordered collections of objects, known as *mapping objects* or *associative arrays*. Unlike lists or strings, dictionaries access values through keys.

```
ani1 = {}
ani1["name"] = "giraffe"
ani1["size"] = 5.0
ani1["weight"] = 1100
print(ani1)
{'name': 'giraffe', 'size': 5.0, 'weight': 1100}
```

- Dictionaries allow the association of multiple keys with corresponding values.
- The Order of elements in dictionaries became consistent in Python 3.7.

Dictionaries

Initialization

Dictionaries can be initialized in one operation:

```
ani2 = {"name": "monkey", "weight": 70, "size": 1.75}
```

Additional keys and values can be added later.

Accessing Values

To retrieve the value associated with a given key:

```
monkey_size = ani1["size"]
```

Advantages

- Human-readable code due to accessing elements by names (keys).
- Improves code readability compared to using numerical indices.

Dictionary Methods

Methods `.keys()`, `.values()`, and `.items()`

The `.keys()` and `.values()` methods return the keys and values of a dictionary, respectively.

```
ani2.keys()
dict_keys(['weight', 'name', 'size'])
ani2.values()
dict_values([70, 'monkey', 1.75])
```

The types `dict_keys` and `dict_values` are not indexable. To convert them to a list, use `list()`.

```
list(ani2.values())
['monkey', 70, 1.75]
```

List of Dictionaries

List of Dictionaries

Creating a list of dictionaries with the same keys resembles a database structure:

```
animals = [ani1, ani2]
animals
[{'name': 'giraffe', 'weight': 1100, 'size': 5.0},
{'name': 'monkey', 'weight': 70, 'size': 1.75}]
```

Dictionaries allow managing complex structures more explicitly than lists.

Tuples

Tuples Definition

Tuples are sequential objects similar to lists (iterable, ordered, and indexable) but are non-modifiable. Their advantage over lists lies in their immutability, which significantly speeds up access and consumes less memory.

```
t = (1, 2, 3)
t
(1, 2, 3)
type(t)
<class 'tuple'>
t[2]
3
t[0:2]
(1, 2)
t[2] = 15
```

Raises a TypeError: 'tuple' object does not support item assignment

Tuples

Tuple Modification

Attempting to modify a tuple element raises a TypeError. To add or modify elements, create a new tuple.

```
t = (1, 2, 3)
id(t)
139971081704464
t = t + (2,)
t
(1, 2, 3, 2)
id(t)
139971081700368
```

More Tuple Features

Tuple Operations

The operators + and * work similarly to lists (concatenation and duplication).

```
(1, 2) + (3, 4)
(1, 2, 3, 4)
(1, 2) * 4
(1, 2, 1, 2, 1, 2, 1, 2)
```

Additionally, the `tuple(sequence)` function functions like `list()`, casting a container object to a tuple.

```
tuple([1, 2, 3])
(1, 2, 3)
```

Remark

Lists, dictionaries, and tuples are containers, i.e., objects containing a collection of other objects. In Python, you can build lists that contain dictionaries, tuples, or other lists, as well as dictionaries that contain tuples, lists, and so on. The combinations are endless!

Table of Contents

- Introduction
  - Computers
  - Algorithms
  - Why Python?
  - Python
- Variables and Data Types
  - Variables
  - Data types
  - Containers
- Loop
  - For Loops
  - While Loops
- Control Structures
- Functions
- Moudule
- File Handling

For Loops

Why Loops?

- In programming, it's often necessary to repeat an instruction multiple times.
- Loops, essential in programming languages, help us perform repetitive tasks concisely and efficiently.
- For example, suppose you want to display the elements of a list one after the other.
- Without loops, you would have to write something like:

```
animals = ["giraffe", "tiger", "monkey", "mouse"]
print(animals[0])
print(animals[1])
print(animals[2])
print(animals[3])
```

- If your list contains only 4 elements, this is manageable, but imagine having 100 or even 1000 elements?

For Loops

Principle of For Loops

To address the previus example, we need to use loops. Consider the following example:

```
animals = ["giraffe", "tiger", "monkey", "mouse"]
for animal in animals:
        print(animal)
```

Explanation

In this example:

- The variable `animal` is called the iteration variable. It takes on the successive values of the list `animals` in each iteration of the loop. (you can choose any name for this variable.)
- The iteration variable `animal` is created by Python the first time the line with `for` is executed.
- Once the loop is finished, this iteration variable `animal` is not destroyed and retains the last value of the list `animals` (in this case, the string "mouse").
- The `animals` variable is a list being iterated over, and `animal` is a string because each element of the list `animals` is a string.

For Loops: Indentation and Examples

Indentation and Structure

- To address the previus example, we need to use loops. Consider the following example:

```
for animal in animals:
        print(animal)
        print(animal * 2)
print("Done")
```

- Line 4 is not part of the loop body as it is at the same level as `for` (not indented relative to `for`).
- Note that each instruction in the loop body must be indented similarly (here, 4 spaces).
- **Note:** While indentation can be done using spaces or tabs (not a mix of both)

For Loops: Indentation and Examples

Indentation and Structure

- Forgetting indentation in Python results in an error message: **IndentationError: expected an indented block**
- In the previous examples, we executed a loop by iterating directly over a list. Since a slice of a list is also a list, we can iterate over it:

```
animals = ["giraffe", "tiger", "monkey", "mouse"]
for animal in animaux[1:3]:
        print(animal)
```

- We've seen that `for` loops can iterate over a list containing strings, but they can also iterate over lists containing integers (or any variable type).

```
for i in [1, 2, 3]:
        print(i)
```

For Loops and range()

Using range() in a For Loop

Python has the `range()` function, which we encountered earlier in Lists, and is convenient for automatically looping over a range of integers:

```
for i in range(4):
    print(i)
```

- Unlike creating a list with `list(range(4))`, the `range()` function can be used directly in a loop. There's no need to type `for i in list(range(4)):`, even though it would work.
- The statement `list(range(4))` simply converts an object of type `range` into an object of type `list`. As a reminder, this casting function converts one type to another. (Python would first construct a list of 4 elements in memory and then iterate over it, leading to a huge waste of time!).

While Loops

Using `while` Loops

An alternative to the commonly used `for` loop in programming is the `while` loop. This type of loop executes a series of instructions as long as a condition is true. For example:

```
1 i = 1
2 while i <= 4:
3         print(i)
4         i = i + 1
```

- Notice that it is necessary to indent the block of instructions corresponding to the body of the loop (here, instructions on lines 3 and 4).
- A `while` loop generally requires three elements to function correctly:
  - Initialization of the iteration variable before the loop (line 1).
  - Testing the iteration variable associated with the `while` statement (line 2).
  - Updating the iteration variable in the loop's body (line 4).

While Loops

Using `while` Loops

- Be careful with your tests and increments because an error often leads to "infinite loops" that never stop. However, you can always stop the execution of a Python script using the Ctrl-C. For example:

```
i = 0
while i < 10:
        print("Python is cool!")
```

- We forgot to update the variable `i` in the loop's body. Consequently, the loop will never stop (except by pressing Ctrl-C) since the condition `i < 10` will always be true.

Table of Contents

- Introduction
  - Computers
  - Algorithms
  - Why Python?
  - Python
- Variables and Data Types
  - Variables
  - Data types
  - Containers
- Loop
  - For Loops
  - While Loops
- Control Structures
- Functions
- Moudule
- File Handling

Tests

Definition

Tests are essential in any computer language if you want to give it a bit of complexity because they allow the computer to make decisions. Python uses the `if` statement and comparison we covered in the previous slides. Here is a first example:

```
x = 2
if x == 2:
        print("The test is true!")
```

And a second one:

```
x = "mouse"
if x == "tiger":
        print("The test is true!")
```

There are several remarks to make regarding these two examples:

- In the first example, the test is true, so the `print("The test is true!")` statement is executed. In the second example, the test is false, and nothing is printed.
- Instruction blocks in tests must be indented, just like in `for` and `while` loops. Indentation indicates the scope of the instructions if the test is true.
- As with `for` and `while` loops, the line containing the `if` statement ends with a ":" character.

Multiple Case Tests

Definition

Sometimes, it is convenient to test whether a condition is true or false in the same `if` statement. Rather than using two `if` statements, you can use the `if` and `else` statements:

```
x = 2
if x == 2:
        print("The test is true!")
else:
        print("The test is False!")

x = 3
if x == 2:
        print("The test is true!")
else:
        print("The test is False!")
```

In these examples, if the condition is true, the statement in the `if` block is executed; otherwise, the statement in the `else` block is executed.

Multiple Case Tests

Example: Randomly Choosing a DNA Base

Let's consider the example of randomly selecting a DNA base and printing its name. In the following code, we use the `random.choice(list)` statement returns a randomly chosen element from a list. The `import random` statement will be covered later, for now, assume that it is necessary.

```
import random
base = random.choice(["a", "t", "c", "g"])
if base == "a":
        print("Choice of an adenine")
elif base == "t":
        print("Choice of a thymine")
elif base == "c":
        print("Choice of a cytosine")
elif base == "g":
        print("Choice of a guanine")
```

In this example, Python tests the first condition, and if and only if it is false, it tests the second, and so on.

Logical Operators in Python

Logical Operators: `and`, `or`, `not`

In Python, the reserved word `and` is used for the AND operator, and the reserved word `or` is used for the OR operator. Pay attention to the case sensitivity of the operators `and` and `or`, written in lowercase in Python. Here is an example of usage:
Example:

```
x = 2
y = 2
if x == 2 and y == 2:
        print("The test is true")
```

Note that the same result would be obtained using two nested `if` statements:
Example:

```
x = 2
y = 2
if x == 2:
if y == 2:
print("The test is true")
```

Logical Operators in Python

Logical Operators: `and`, `or`, `not`

You can use the logical NOT operator (`not`) to invert the result of a condition: **Example:**

```
not True
# Output: False
not False
# Output: True
not (True and True)
# Output: False
```

break and continue Statements

break Statement

The `break` statement is used to exit a loop (either `for` or `while`) prematurely. It stops the loop before it has completed all its iterations.

Example:

```
for i in range(5):
        print(i)
        if i > 2:
                break
# Output:
# 0
# 1
# 2
```

break and continue Statements

continue Statement

The continue statement is used to skip the rest of the code inside a loop for the current iteration and proceed to the next iteration. **Example:**

```
for i in range(5):
        if i == 2:
                continue
        print(i)
# Output:
# 0
# 1
# 3
# 4
```

Table of Contents

- Introduction
  - Computers
  - Algorithms
  - Why Python?
  - Python
- Variables and Data Types
  - Variables
  - Data types
  - Containers
- Loop
  - For Loops
  - While Loops
- Control Structures
- Functions
- Moudule
- File Handling

Functions in Python

Introduction

In programming, functions are essential for performing the same operation multiple times within a program. They also make the code more readable and clear by breaking it down into logical blocks.
You're already familiar with some Python built-in functions like `range()` or `len()`.
For now, think of a function as a kind of "black box":

- You pass zero, one, or more variables (arguments) inside parentheses. These variables can be of any Python data type.
- It performs an action.
- It returns a Python object or nothing.

For example, when you call the `len()` function like this:

```
>>> len([0, 1, 2])
3
```

Here's what happens:

- You call `len()` with a list as an argument (here, the list $[0, 1, 2]$).
- The function calculates the length of this list.
- It returns an integer equal to this length.

Functions in Python

From the Programmer's Perspective

From the programmer's perspective, a function is a portion of code that performs a specific sequence of instructions. However, let's revisit the concept of a "black box" one last time:

- A function performs a task, potentially receiving arguments and returning something. The algorithm used inside the function is not directly relevant to the user. For example, it's unnecessary to know how the `len` function calculates the length of a list. It requires a list as an argument and returns its length. What happens inside the function is the concern of the programmer.
- Each function typically performs a single, specific task. Writing multiple functions (which can call each other) is better if things get complicated. This modularity improves the overall quality and readability of the code.

- In general, we used the term "main program" to refer to where a function is called (we'll see later that a function can be called from anywhere).
- The main program denotes the code that is executed when launching the Python script, i.e., all the instructions outside of functions.
- Generally, in a Python script, functions are defined first, followed by the main program.

Defining Functions in Python

Defining a Function

The `def` keyword defines a function in Python. If the function is expected to return a value, the `return` keyword is included. Here's an example:

```
def square(x):
return x ** 2
```

The `def` syntax, like `for` and `while` loops or `if` statements, uses ":" to indicate the beginning of a block of instructions, known as the function body. Proper indentation of this block is mandatory.
In the given example, the `square()` function takes an argument `x` and returns its square. This returned value can be stored in a variable:

```
result = square(2)
print(result)
# Output: 4
```

Functions Without Arguments and Return Values

Functions Without Arguments and Return Values

A function in Python doesn't necessarily need to take an argument or return a value. For example:

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

In this case, the `hello()` function prints the string "Hello" to the screen. It takes no arguments and returns nothing. Therefore, trying to retrieve a returned value from such a function doesn't make sense. If attempted, Python assigns the value `None`, which means nothing in English:

```
result = hello()
print(result)  # Output: None
```

This is not an error since Python does not raise an error. However, it is generally of no interest.

Variable Number of Arguments

Variable Number of Arguments

The number of arguments that can be passed to a function is variable. You encountered functions that took 0 or 1. You also saw internal Python functions that took at least two arguments, such as `range(1, 10)` or `range(1, 10, 2)`. The number of arguments is left to the discretion of the programmer developing a new function.
A particularity of functions in Python is that you are not obliged to specify the type of arguments you pass to it as long as the operations you perform with these arguments are valid.
For example:

```
def multiply(x, y):
return x * y

print(multiply(2, 3))           # Output: 6
print(multiply(3.1415, 5.23))   # Output: 16.430045000000003
print(multiply("to", 2))        # Output: 'toto'
print(multiply([1, 3], 2))       # Output: [1, 3, 1, 3]
```

The * operator recognizes multiple types (integers, floats, strings, lists). Our `multiply()` function can perform different tasks! Although Python allows this, be cautious of this great flexibility that could lead to surprises in your future programs. Generally, it is wiser for each argument to have a specific type (integers, floats, strings, etc.).

Returning Multiple Results

Returning Multiple Results

A huge advantage in Python is that functions can return multiple objects at once. Example:

```
def square_cube(x):
return x**2, x**3

result = square_cube(2)
print(result)  # Output: (4, 8)
```

In reality, Python only returns a single object, but this object can be sequential, meaning it can contain other objects. In our example, Python returns an object of type tuple.

```
def square_cube_2(x):
return [x**2, x**3]

result = square_cube_2(3)
print(result)  # Output: [9, 27]
```

Returning a tuple of two (or more) elements is very convenient in conjunction with multiple assignments, for example:

```
z1, z2 = square_cube_2(3)
print(z1)  # Output: 9
print(z2)  # Output: 27
```

Positional Arguments

Positional Arguments

So far, we have consistently passed the number of arguments the function expected. What happens if a function expects two arguments and we only pass one?

```
def multiply(x, y):
return x * y

result_1 = multiply(2, 3)
print(result_1)  # Output: 6

result_2 = multiply(2)
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# TypeError: multiply() missing 1 required
# positional argument: 'y'
```

It is passing only one argument to a function that expects two leads to an error.
Definition: When defining a function `def func(x, y):`, the arguments `x` and `y` are called **positional arguments**. It is strictly necessary to specify them when calling the function. Moreover, it is necessary to respect the same order when calling as in the function definition. In the example above, 2 will correspond to `x`, and 3 will correspond to `y`.

Default and Keyword Arguments

Default Arguments and Keyword Arguments

It is possible to pass one or more arguments optionally and assign them default values:

```
def func(x=1):
return x

result_1 = func()
print(result_1)  # Output: 1

result_2 = func(10)
print(result_2)  # Output: 10
```

Definition: An argument defined with the syntax `def func(arg=val):` is called a **keyword argument**. The passage of such an argument when calling the function is optional. This type of argument should not be confused with positional arguments presented above, whose syntax is `def func(arg):`.

Default and Keyword Arguments

Default Arguments and Keyword Arguments

It is, of course, possible to pass several keyword arguments:

```
def func(x=0, y=0, z=0):
return x, y, z

result_1 = func()
print(result_1)  # Output: (0, 0, 0)

result_2 = func(10)
print(result_2)  # Output: (10, 0, 0)

result_3 = func(10, 8)
print(result_3)  # Output: (10, 8, 0)

result_4 = func(10, 8, 3)
print(result_4)  # Output: (10, 8, 3)
```

Default and Keyword Arguments

Default Arguments and Keyword Arguments

- Python allows entering keyword arguments in an arbitrary order as long as their name is specified.
- If there is a mix of positional and keyword arguments, positional arguments must always be placed before keyword arguments.

```
def func(a, b, x=0, y=0, z=0):
return a, b, x, y, z

result_1 = func(1, 1)
print(result_1)  # Output: (1, 1, 0, 0, 0)

result_2 = func(1, 1, z=5)
print(result_2)  # Output: (1, 1, 0, 0, 5)

result_3 = func(1, 1, z=5, y=32)
print(result_3)  # Output: (1, 1, 0, 32, 5)
```

Default and Keyword Arguments

Passing keyword arguments in an arbitrary order is allowed as long as their name is specified. However, if the positional arguments `a` and `b` are not passed to the function, Python raises an error.

```
result_4 = func(z=0)
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# TypeError: func() missing 2 required
# positional arguments: 'a' and 'b'
```

Tip: Specifying the names of keyword arguments when calling a function is a recommended practice. This clearly distinguishes them from positional arguments.

Local and Global Variables

- When working with functions, it's crucial to understand how variables behave.
- A variable is termed *local* when created within a function. It exists and is visible only during the execution of that function.
- A variable is termed *global* when created in the main program. It is visible throughout the entire program.
- Example:

```
# Definition of a function square()
def square(x):
y = x ** 2
return y

# Main program
z = 5
result = square(z)
print(result)
```

Execution Steps

Execution Steps - Understanding the Code

Let's now examine what happens in the above code step by step:

- **Step 1:** Python is ready to read the first line of code.
- **Step 2:** Python stores the `square()` function in memory. Note that it does not execute it! The function is placed in a space in memory, which is the space of the main program. All global variables created in the program will be stored in this space. Python is now ready to execute the main program.
- **Step 3:** Python reads and stores the variable `z`. It will be a global variable since it is created in the main program. Thus, it will also be stored in the Global frame.
- **Step 4:** The `square()` function is called, and the integer `z` is passed as an argument. The function executes. Note that the variable passed as an argument, named `x` in the function, is created as a local variable.
- **Step 5:** Python is now ready to execute each line of code in the function.
- **Step 6:** The variable `y` is created in the function. This is stored as a local variable to the function.
- **Step 7:** Python is about to return the local variable `y` to the main program.
- **Step 8:** Python exits the function, and the value it returns is assigned to the global variable `result`. Note that the space for the function's variables is destroyed when Python exits the function. Thus, all variables created in the function no longer exist. This is why they are called local since they only exist when the function is executed.
- **Step 9:** Python displays the content of the variable `result`, and the execution is complete.

Calling Functions Within Functions

Calling Functions Within Functions

It is possible to call a function from another function. More generally, a function can be called from anywhere as long as it is visible to Python. Consider the following example:

```
# Function definitions
def polynomial(x):
return x**2 - 2*x + 1

def calculate_values(start, end):
value_list = []
for x in range(start, end + 1):
value_list.append(polynomial(x))
return value_list

# Main program
print(calculate_values(-5, 5))
```

- Here, we call the `calculate_values()` function from the main program, and within it, we call the `polynomial()` function. Let's see what happens when executing the `polynomial()` function.
- Thus, the programmer is free to make as many calls as desired. A function can call another function, and that function can call another function, and so on (as many times as needed).

Recursive Functions

Recursive Functions

In Python, a function can call itself, known as a **recursive function**.
Let's consider a classic example: calculating the factorial of a number.

```
def factorial(n):
if n == 0 or n == 1:
return 1
else:
return n * factorial(n - 1)

result = factorial(5)
print(result)  # Output: 120
```

Here, the `factorial()` function calls itself with a smaller argument (`n - 1`) until it reaches the base case (`n == 0` or `n == 1`). This is a common pattern in recursive functions.

Variable Scope

Variable Scope

When working with functions, it is crucial to understand the scope of variables—where they are visible. We have seen that variables created within a function are not visible outside of it because they are local to the function. Consider the following code:

```
def my_function():
    x = 2
    print("x = ", x, "within the function")

my_function()
# Output: x = 2 within the function

print(x)
# Error: NameError: name 'x' is not defined
```

- When Python executes the function code, it knows the content of the variable x. However, back in the main module (in this case, the Python interpreter), it no longer knows it, resulting in an error.
- Similarly, a variable passed as an argument is considered local when inside the function.

Variable Scope

Variable Scope in Main Program and Functions

When a variable is declared in the main program, it is visible in the program and in all functions. We refer to this as a global variable:

```
def my_function():
        print(x)

x = 3
my_function()
# Output: 3

print(x)
# Output: 3
```

In this case, the variable x is visible in the main module and in all functions of the module.

Variable Scope

Variable Scope in Main Program and Functions

Python does not allow the modification of a global variable in a function:

```
def my_function():
x = x + 1

x = 1
my_function()
# Error: UnboundLocalError: local variable
# 'x' referenced before assignment
```

The error indicates that Python considers x as a local variable that has not been assigned yet. To truly modify a global variable in a function, the `global` keyword must be used:

```
def my_function():
global x
x = x + 1

x = 1
my_function()
print(x)
# Output: 2
```

Mutable Types in Functions

Caution with Mutable Types (e.g., Lists)

Be extremely careful with mutable types (such as lists) because you can modify them within a function:

```
def my_function():
        my_list[1] = -127

my_list = [1, 2, 3]
my_function()
print(my_list)
# Output: [1, -127, 3]
```

Similarly, if you pass a list as an argument, it can be modified within the function:

```
def my_function(x):
        x[1] = -15

my_list = [1, 2, 3]
my_function(my_list)
print(my_list)
# Output: [1, -15, 3]
```

If you want to avoid modifying a list in a function, use tuples or explicitly pass a copy to the function.

Modifying a List in a Function

Best Practice: Indicate List Modification

When dealing with functions that modify a list, it is advisable to clearly indicate this behavior in your code. Make the function return the modified list and capture this returned list in a variable with the same name. For example:

```
def add_one(lst):
        for index in range(len(lst)):
                lst[index] += 1
        return lst

# Main program
grade_list = [10, 8, 16, 7, 15]
grade_list = add_one(grade_list)
print(grade_list)
```

The line 8 indicates that the `grade_list` passed to the function is replaced by the list returned from the function.

Modifying a List in a Function

Best Practice: Indicate List Modification

Alternatively, the following code would produce the same output:

```
def add_one(lst):
for index in range(len(lst)):
lst[index] += 1

# Main program
grade_list = [10, 8, 16, 7, 15]
add_one(grade_list)
print(grade_list)
```

However, this is less intuitive since it may not be evident that the list is modified within the function by reading line 7. In such cases, it is crucial to document in the function's documentation that the list is modified "in place."

Tip: Instead of using global variables, explicitly pass your variables to functions as argument(s).

Why Create Your Own Modules?

Module Creation in Python

A well-written function could be reused in another Python program. This is precisely the purpose of creating a module – a collection of functions often used together. Generally, modules are organized around a specific theme.

Creating a Module in Python

Simple Example: message.py

```
"""A useless module that displays messages :-)."""

DATE = 16092008

def say_hello(name):
"""Says Hello."""
return "Hello " + name

def say_goodbye(name):
"""Says Goodbye."""
return "Goodbye " + name

def say_greetings(name):
"""Says Greetings."""
return "Greetings " + name
```

The triple-quoted strings at the beginning of the module and each function are optional but play an essential role in code documentation.

Using Your Own Module

Module Utilization in Python

To call a function or variable from the module, the message.py file must be in the current directory. After that, you can **import** the module, and all its functions (and constants) become accessible.

Importing and Using the Module

Load the module with the command `import message`. Note that the file is saved with a .py extension, yet we don't specify it when importing the module. After that, you can use the functions as with a regular module.

```
import message
message.say_hello("Joe")    # Output: 'Hello Joe'
message.say_goodbye("Mari")  # Output: 'Goodbye Mari'
message.say_greetings("Sir")  # Output: 'Greetings Sir'
message.DATE  # Output: 16092008
```

Docstrings

Importance of Documentation in Modules

When writing a module, it's crucial to create documentation explaining what it does and how to use each function. The triple-quoted strings at the beginning of the module and each function serve this purpose and are called docstrings.

Generating Help with docstrings

Generating Help with docstrings

Docstrings provide help when invoking the `help()` command:

help (message)

This generates an automatic help page for the entire module, including functions and data.

Contents of a Good Docstring

Components of a Good Docstring

In summary, a good function docstring should contain:

- What the function does,
- What arguments it takes,
- What it returns.

Docstrings are for module users and serve a different purpose than comments, which are for code readers.

Visibility of Functions in a Module

Visibility Rules in Modules

The visibility of functions within modules follows simple rules:

- Functions within the same module can call each other.
- Functions within a module can call functions in another module if it has been previously imported. For example, if the command `import anothermodule` is used in a module, it is possible to call a function with `anothermodule.function()`.

Executing Scripts as Modules

Structure for Script/Module Compatibility

When creating Python code that can be used both as a script and as a module, it's advisable to follow this structure:

```
"""Test script."""

def Hello(name):
"""Say Hello."""
return "Hello " + name

if __name__ == "__main__":
        print(Hello("Joe"))
```

The condition `if __name__ == "__main__":` allows the code to behave differently when executed as a script or imported as a module. This structure enhances code readability and eliminates the need for additional comments.

Reading from a File

Method `.readlines()`

- Much scientific information is stored as text in files.
- Python provides tools to simplify reading or writing to one or more files.

Example

```
# Create a file named zoo.txt with the content:
# girrafe
# tiger
# monkey
# mouse

# Test the following code in the Python interpreter:

filin = open("zoo.txt", "r")  # Open the file in read mode
print(filin)  # Display the file object

lines = filin.readlines()  # Read all lines from the file
print(lines)  # Display the lines
filin.close()  # Close the file

# Attempting to read from a closed file will result in an error
# filin.readlines()  # ValueError: I/O operation on closed file
```

Reading from a File

- The `open()` function is used to open the file in read mode ("r").
- The `readlines()` method reads all lines from the file, and the resulting list contains newline characters.
- The `close()` method is used to close the file.
- Attempting to read from a closed file raises a `ValueError`.

Example: Reading a File with Python

Complete File Reading Example

- Here is a complete example of reading a file in Python:

```
# Open the file zoo.txt in read mode
filin = open("zoo.txt", "r")

# Read all lines from the file and store them in a list
lines = filin.readlines()

# Display the lines
print(lines)

# Iterate through the lines and print each line
for line in lines:
        print(line)

# Close the file
filin.close()
```

- The code opens the file, reads all lines, displays them, iterates through each line, prints them, and finally closes the file.

Note on File Reading Example

Observations

- Each element in the `lines` list is a string. Python reads the content of a file as strings.
- Each element in the `lines` list ends with the `\n` character, representing a "newline" or "line feed."

Efficient File Handling with `with`

Using `with` Statement

In Python, the `with` keyword ensures efficient handling of file operations, automatically closing the file even if an error occurs during opening or reading. The same file reading example can be implemented using `with` as follows:

```
with open("zoo.txt", 'r') as filin:
        lines = filin.readlines()
        for line in lines:
                print(line)
```

Reading the Entire File with .read()

Using .read() Method

The .read() method reads the entire content of a file and returns a single string. Here's an example:

```
with open("zoo.txt", "r") as filin:
    content = filin.read()
    print(content)
```

Reading a File Line by Line with .readline()

Using .readline() Method

The .readline() method reads a line from a file and returns it as a string. With a while loop, you can read the file line by line:

```
with open("zoo.txt", "r") as filin:
        line = filin.readline()
        while line != "":
                print(line)
                line = filin.readline()
```

Iterating Directly Over a File in Python

Iterating Directly Over a File

Python provides a convenient way to iterate directly over the lines of a file:

```
with open("zoo.txt", "r") as filin:
    for line in filin:
        print(line)
```

Writing to a File in Python

Writing to a File

Writing to a file in Python is as straightforward as reading from it. Here's an example:

```
animals2 = ["fish", "dog", "cat"]

with open("zoo2.txt", "w") as filout:
        for animal in animals:
                filout.write(f"{animal}\n")
```

This code writes each element of the list `animals2` to the file `zoo2.txt`, with each element on a new line.

Opening Two Files with `with` Statement

Opening Two Files

The `with` statement in Python allows you to open and work with multiple files simultaneously. Here's an example:

```
with open("zoo.txt", "r") as file1, open("zoo2.txt", "w") as fil
for line in file1:
        file2.write("* " + line)
```

This code opens two files, `zoo.txt` for reading and `zoo2.txt` for writing. It reads each line from `file1` and writes a modified version to `file2`.

Importance of Type Conversions with Files

Type Conversions with Files

When reading from or writing to files in Python, it's crucial to consider those file methods, such as `.readlines()`, always return strings. Similarly, when writing to a file, you must provide a string to the `.write()` method.
To handle these constraints, you'll need to use the type conversion functions introduced previously, `float()`, and `str()`).
These conversion functions are essential, especially when dealing with file numerical data. Numbers in a file are treated as text (strings), so you must convert them to integers or floats to perform numerical operations.