Python Essentials 2: INTERMEDIATE

Dec 2021

Content

[Module 1: Modules, Packages and PIP 12/26/21 10](#_Toc28824)

[1.1 INTRODUCTION TO MODULES IN PYTHON 10](#_Toc4589)

[1.1.1.1 Modules: What is a module? 10](#_Toc21615)

[1.1.1.2 Using Modules: How to make use of a module? 11](#_Toc27225)

[1.1.1.3 Using Modules: Importing a module 11](#_Toc23928)

[1.1.1.4 Using Modules: Importing a module continued 12](#_Toc15864)

[1.1.1.5 Importing a module | math 13](#_Toc9613)

[1.1.1.6 Importing a module | math 13](#_Toc28540)

[1.1.1.7 Importing a module | math 13](#_Toc31704)

[1.1.1.8 Importing a module | math 14](#_Toc5052)

[1.1.1.9 Importing a module | \* and as 15](#_Toc25303)

[1.1.1.10 Importing a module | aliasing 15](#_Toc2076)

[1.1.1.11 SECTION SUMMARY: Key takeaways 16](#_Toc7102)

[1.2. SELECTED PYTHON MODULES (MATH, RANDOM, PLATFORM) 17](#_Toc25846)

[1.2.1.1 Useful Modules 17](#_Toc4517)

[1.2.1.2 Useful modules | math 18](#_Toc29826)

[1.2.1.3 Useful modules | math 19](#_Toc19297)

[1.2.1.4 Useful modules | math 20](#_Toc22994)

[1.2.1.5 Useful modules | random 20](#_Toc257)

[1.2.1.6 Useful modules | random 21](#_Toc12510)

[1.2.1.7 Useful modules | random 22](#_Toc6716)

[1.2.1.8 Useful modules | random 22](#_Toc30641)

[1.2.1.9 Useful modules | platform 12/27/21 23](#_Toc17755)

[1.2.1.10 Useful modules | platform: platform function 24](#_Toc21041)

[1.2.1.11 Useful modules | platform: machine function 25](#_Toc4705)

[1.2.1.12 Useful modules | platform: processor function 25](#_Toc1320)

[1.2.1.13 Useful modules | platform: system function 26](#_Toc7294)

[1.2.1.14 Useful modules | platform: version function 26](#_Toc10858)

[1.2.1.15 Useful modules | platform: 2 more functions 26](#_Toc23568)

[1.2.1.16 Useful modules | Python Module Index 27](#_Toc28492)

[1.2.1.17 SECTION SUMMARY: Key takeaways 27](#_Toc19906)

[1.3. MODULES AND PACKAGES 28](#_Toc30671)

[1.3.1.1 Modules and Packages: What is a package? 28](#_Toc16264)

[1.3.1.2 Modules and Packages: Your first module: step 1-2 28](#_Toc5165)

[1.3.1.3 Modules and Packages: Your first module: step 3-6 29](#_Toc23171)

[1.3.1.4 Modules and Packages: Your first module: step 7-10 12/28/21 31](#_Toc9119)

[1.3.1.5 Modules and Packages: Your first module: step 11-12 32](#_Toc31005)

[1.3.1.6 Modules and Packages: step 1-2 33](#_Toc5792)

[1.3.1.7 Modules and Packages: step 3-4 35](#_Toc4819)

[1.3.1.8 Modules and Packages: step 5-6 35](#_Toc21494)

[1.3.1.9 Modules and Packages: step 7 36](#_Toc5003)

[1.3.1.10 Modules and Packages: step 8-9 37](#_Toc8002)

[1.3.1.11 SECTION SUMMARY 38](#_Toc25196)

[1.4. PYTHON PACKAGE INSTALLER (PIP) 39](#_Toc21744)

[1.4.1.1 Python Package Installer (PIP): packaging ecosystem 39](#_Toc15498)

[1.4.1.2 Python Package Installer (PIP): packaging ecosystem 39](#_Toc11961)

[1.4.1.3 Python Package Installer (PIP): The PyPI repo 40](#_Toc27112)

[1.4.1.4 Python Package Installer (PIP): The PyPI repo 40](#_Toc7441)

[1.4.1.5 Python Package Installer (PIP): How to install pip 41](#_Toc18338)

[1.4.1.6 pip installation on MS Windows 41](#_Toc25917)

[1.4.1.7 pip installation on Linux 41](#_Toc14389)

[1.4.1.8 pip installation on Linux 42](#_Toc28901)

[1.4.1.9 pip installation on Linux 43](#_Toc21313)

[1.4.1.10 Dependencies 44](#_Toc25432)

[1.4.1.11 How to use pip: How to use pip 44](#_Toc14403)

[1.4.1.12 How to use pip: How to use pip: continued 45](#_Toc27107)

[1.4.1.13 How to use pip: How to use pip: continued 46](#_Toc17419)

[1.4.1.14 How to use pip: How to use pip: a simple test program 47](#_Toc10217)

[1.4.1.15 How to use pip: How to use pip: continued 48](#_Toc9983)

[1.4.1.16 How to use pip: How to use pip: continued 49](#_Toc25751)

[1.4.1.17 How to use pip: Use pip! 49](#_Toc2673)

[1.4.1.18 SECTION SUMMARY: Key takeaways 50](#_Toc3951)

[1.4.1.19 Module Completion 50](#_Toc9605)

[1.5. PE2 -- Module 1 Quiz 51](#_Toc6657)

[Module 1 Test 12/29/21 53](#_Toc26245)

[MODULE 2: STRINGS, STRING AND LIST METHODS, AND EXCEPTIONS 57](#_Toc22660)

[2.1. CHARACTERS, STRINGS, COMPUTERS 57](#_Toc15249)

[2.1.1.0 Python Essentials 2 - Module 2 57](#_Toc26680)

[2.1.1.1 Characters and Strings vs. Computers: ASCII 57](#_Toc7692)

[2.1.1.2 Characters and Strings vs. Computers: I18N 58](#_Toc20904)

[2.1.1.3 Characters and Strings vs. Computers: Unicode, UCS-4,UTF-8 59](#_Toc24119)

[2.1.1.4 SECTION SUMMARY 60](#_Toc3666)

[2.2. THE NATURE OF STRINGS IN PYTHON 61](#_Toc22741)

[2.2.1.1 The nature of strings in Python: Strings - a brief review 61](#_Toc5184)

[2.2.1.2 The nature of strings in Python: Multiline strings 61](#_Toc20181)

[2.2.1.3 The nature of strings in Python: Operations on strings 62](#_Toc24456)

[2.2.1.4 The nature of strings in Python: Operations on strings: ord() 62](#_Toc5962)

[2.2.1.5 The nature of strings in Python: Operations on strings: chr() 63](#_Toc28894)

[2.2.1.6 The nature of strings in Python: Strings as sequences: indexing 63](#_Toc26181)

[2.2.1.7 The nature of strings in Python: Slices 64](#_Toc12694)

[2.2.1.8 The nature of strings in Python: The in and not in operators 64](#_Toc30533)

[2.2.1.9 The nature of strings in Python: Python strings are immutable 65](#_Toc28867)

[2.2.1.10 The nature of strings in Python: Operations on strings: continued 65](#_Toc8279)

[2.2.1.11 The nature of strings in Python: Operations on strings: min() 65](#_Toc2657)

[2.2.1.12 The nature of strings in Python: Operations on strings: max() 66](#_Toc15141)

[2.2.1.13 The nature of strings in Python: the index() method 66](#_Toc4259)

[2.2.1.14 The nature of strings in Python: Operations on strings: the list() function 67](#_Toc31614)

[2.2.1.15 SECTION SUMMARY 67](#_Toc20700)

[2.3. STRING METHODS 68](#_Toc18208)

[2.3.1.1 String methods: The capitalize() method 68](#_Toc6312)

[2.3.1.2 String methods: The center() method 69](#_Toc201)

[2.3.1.3 String methods: The endswith() method 70](#_Toc1324)

[2.3.1.4 String methods: The find() method 70](#_Toc2267)

[2.3.1.5 String methods: The isalnum() method 71](#_Toc13753)

[2.3.1.6 String methods: The isalpha() and isdigit() method 72](#_Toc2069)

[2.3.1.7 String methods: The islower(), isspace(), isupper() method 72](#_Toc32055)

[2.3.1.8 String methods: The join() method 73](#_Toc14331)

[2.3.1.9 String methods: The lower() method 73](#_Toc3202)

[2.3.1.10 String methods: The lstrip() method 73](#_Toc6506)

[2.3.1.11 String methods: The replace() method 12/30/21 74](#_Toc18542)

[2.3.1.12 String methods: The rfind() method 74](#_Toc6808)

[2.3.1.13 String methods: The rstrip() method 74](#_Toc9507)

[2.3.1.14 String methods: The split() method 75](#_Toc26202)

[2.3.1.15 String methods: The startswith() and strip() method 75](#_Toc5216)

[2.3.1.16 String methods: The swapcase(), title(), upper() method 75](#_Toc14528)

[2.3.1.17 SECTION SUMMARY 76](#_Toc30202)

[2.3.1.18 Your own split 77](#_Toc24108)

[2.4. STRINGS IN ACTION AND LIST METHODS 78](#_Toc10535)

[2.4.1.1 String in action: Comparing strings 78](#_Toc21486)

[2.4.1.2 String in action: Comparing strings: continued 78](#_Toc11555)

[2.4.1.3 String in action: Sorting 79](#_Toc24279)

[2.4.1.4 String in action: Strings vs. numbers 80](#_Toc29681)

[2.4.1.5 SECTION SUMMARY 80](#_Toc20333)

[2.4.1.6 LAB: A LED Display 81](#_Toc24655)

[2.5. STRINGS AND THE FOUR SIMPLE PROGRAMS 82](#_Toc10395)

[2.5.1.1 Four simple programs:The Caesar Cipher: encrypting a message 82](#_Toc28687)

[2.5.1.2 Four simple programs: The Caesar Cipher: decrypting a message 83](#_Toc17690)

[2.5.1.3 Four simple programs: The Numbers Processor 83](#_Toc24180)

[2.5.1.4 Four simple programs: The IBAN Validator 84](#_Toc14564)

[2.5.1.5 SECTION SUMMARY 86](#_Toc9801)

[2.5.1.6 LAB: Improving the Caesar cipher 12/31/21 87](#_Toc22379)

[2.5.1.7 LAB: Palindromes (To be worked on) 88](#_Toc18883)

[2.5.1.8 LAB: Anagrams (To be worked on) 88](#_Toc19569)

[2.5.1.9 LAB: The Digit of Life(To be worked on) 89](#_Toc13941)

[2.5.1.10 LAB: Find a word!(To be worked on) 89](#_Toc25776)

[2.5.1.11 LAB: Find a word!(To be worked on) 89](#_Toc3547)

[2.6. ERRORS - THE PROGRAMMER'S DAILY BREAD 89](#_Toc19637)

[2.6.1.1 Errors - the programmer's daily bread 89](#_Toc8008)

[2.6.1.2 Errors - the programmer's daily bread: Exceptions 90](#_Toc32665)

[2.6.1.3 Errors - the programmer's daily bread: Exceptions: continued 90](#_Toc1596)

[2.6.1.4 Errors - the programmer's daily bread: Exceptions: continued 91](#_Toc11390)

[2.6.1.5 Errors - the programmer's daily bread: Exceptions: continued 91](#_Toc14443)

[2.6.1.6 Errors - the programmer's daily bread | try-except 91](#_Toc20729)

[2.6.1.7 Errors - the programmer's daily bread | try-except: 92](#_Toc26878)

[2.6.1.8 Errors - the programmer's daily bread | try-except 93](#_Toc30676)

[2.6.1.9 Errors - the programmer's daily bread | try-except 93](#_Toc4817)

[2.6.1.10 Errors - the programmer's daily bread | try-except: 94](#_Toc18121)

[2.6.1.11 Errors - the programmer's daily bread | try-except: 95](#_Toc6948)

[2.6.1.12 Errors - the programmer's daily bread | try-except: 96](#_Toc13129)

[2.7. THE HIERARCHY OF EXCEPTIONS 97](#_Toc13424)

[2.7.1.1 The anatomy of exceptions: Exceptions 97](#_Toc18666)

[2.7.1.2 The anatomy of exceptions: Exceptions: continued 98](#_Toc9630)

[2.7.1.3 The anatomy of exceptions: Exceptions: continued 99](#_Toc23375)

[2.7.1.4 The anatomy of exceptions: Exceptions: continued 1/1/2022 99](#_Toc18498)

[2.7.1.5 The anatomy of exceptions | raise: Exceptions: continued 100](#_Toc17157)

[2.7.1.6 The anatomy of exceptions | raise: Exceptions: continued 101](#_Toc2779)

[2.7.1.7 The anatomy of exceptions | raise: Exceptions: continued 102](#_Toc21544)

[2.7.1.8 SECTION SUMMARY 102](#_Toc31302)

[2.8. USEFUL EXCEPTIONS 104](#_Toc5261)

[2.8.1.1 Useful exceptions: Built-in exceptions: ArithmeticError, AssertionError 104](#_Toc10179)

[2.8.1.2 Useful exceptions: KeyboardInterrupt, LookupError, MemoryError, 105](#_Toc28718)

[2.8.1.3 Useful exceptions: ImportError and KeyError 107](#_Toc22588)

[2.8.1.4 Reading ints safely 108](#_Toc17877)

[2.8.1.5 SECTION SUMMARY 108](#_Toc3758)

[2.8.1.6 Module Completion 109](#_Toc13928)

[2.9 PE2 Module 2 Quiz 1/2/2022 109](#_Toc21214)

[Module 2 Test (with all correct answers) 111](#_Toc13117)

[MODULE 3: Object-Oriented Programming 113](#_Toc24265)

[3.1. THE FOUNDATIONS OF OOP (CLASSES, OBJECTS, ATTRIBUTES) 113](#_Toc27741)

[3.1.1.0 Python Essentials 2 - Module 3 113](#_Toc28775)

[3.1.1.1 The foundations of OOP 114](#_Toc1448)

[3.1.1.2 The foundations of OOP 114](#_Toc10940)

[3.1.1.3 The foundations of OOP: Class hierarchies 115](#_Toc29226)

[3.1.1.4 The foundations of OOP: Class hierarchies continued 116](#_Toc2130)

[3.1.1.5 The foundations of OOP: What is an object? 116](#_Toc3888)

[3.1.1.6 The foundations of OOP: What does an object have? 117](#_Toc12937)

[3.1.1.7 The foundations of OOP: Your first class 118](#_Toc9427)

[3.1.1.8 SECTION SUMMARY 119](#_Toc27748)

[3.2. STACK: THE PROCEDURAL VS OOP APPROACH 120](#_Toc24600)

[3.2.1.1 A short journey from procedural to object approach：What is a stack? 120](#_Toc29879)

[3.2.1.2 A short journey: The stack - the procedural approach 120](#_Toc15063)

[3.2.1.3 The stack - the procedural approach vs. the object-oriented approach 1/3/2022 121](#_Toc8940)

[3.2.1.4 The stack - the object approach 1/4/2022 122](#_Toc10241)

[3.2.1.5 The stack - the object approach: continued 123](#_Toc23217)

[3.2.1.6 The stack - the object approach: continued 123](#_Toc20231)

[3.2.1.7 The object approach: a stack from scratch 124](#_Toc5706)

[3.2.1.8 The object approach: a stack from scratch 125](#_Toc14177)

[3.2.1.9 The object approach: a stack from scratch (continued) 125](#_Toc1129)

[3.2.1.10 The object approach: a stack from scratch (continued) 126](#_Toc3479)

[3.2.1.11 The object approach: a stack from scratch (continued) 1/6/22 127](#_Toc23982)

[3.2.1.12 The object approach: a stack from scratch (continued) 1/7/22 127](#_Toc13618)

[3.2.1.13 SECTION SUMMARY 128](#_Toc30528)

[3.2.1.14 Counting stack (To be worked on) 129](#_Toc14198)

[3.2.1.15 Queue aka FIFO (To be worked on) 129](#_Toc802)

[3.2.1.16 Queue aka FIFO: part 2 (To be worked on) 129](#_Toc23783)

[3.3. PROPERTIES (INSTANCE VAR, CLASS VAR, ATTRIBUTES) 129](#_Toc13354)

[3.3.1.1 OOP: Properties: Instance variables 1/8/22 129](#_Toc28635)

[3.3.1.2 OOP: Properties: Instance variables: continued 130](#_Toc27339)

[3.3.1.3 OOP: Properties: Class variables 131](#_Toc22692)

[3.3.1.4 OOP: Properties: Class variables: continued 132](#_Toc30989)

[3.3.1.5 OOP: Properties: Class variables: continued 132](#_Toc2081)

[3.3.1.6 OOP: Properties: Checking an attribute's existence 133](#_Toc16235)

[3.3.1.7 OOP: Properties: Checking an attribute's existence: continued 134](#_Toc2990)

[3.3.1.8 OOP: Properties: Checking an attribute's existence: continued 135](#_Toc12651)

[3.3.1.9 SECTION SUMMARY 135](#_Toc15828)

[3.4. METHODS (Class and Object Methods, Constructors, Parameters, Properties) 136](#_Toc2091)

[3.4.1.1 OOP: Methods: Methods in detail 136](#_Toc24865)

[3.4.1.2 OOP: Methods in detail: continued 1/9/22 137](#_Toc23105)

[3.4.1.3 OOP: Methods in detail: continued 138](#_Toc7948)

[3.4.1.4 OOP: Methods in detail: continued 138](#_Toc19352)

[3.4.1.5 OOP: Methods: The inner life of classes and objects 1/10/22 139](#_Toc29409)

[3.4.1.6 OOP: Methods: The inner life of classes and objects: Continued 140](#_Toc23855)

[3.4.1.7 OOP: Methods: The inner life of classes and objects: Continued 140](#_Toc2297)

[3.4.1.8 OOP: Methods: The inner life of classes and objects: Continued 140](#_Toc25579)

[3.4.1.8 OOP: Methods: Reflection and introspection 141](#_Toc7378)

[3.4.1.10 OOP: Methods: Investigating classes 141](#_Toc9944)

[3.4.1.11 SECTION SUMMARY 142](#_Toc29888)

[3.4.1.12 The Timer class (To be worked on) 1/11/22 143](#_Toc26765)

[3.4.1.13 Days of the week (To be worked on) 143](#_Toc20556)

[3.4.1.14 Points on a plane (To be worked on) 143](#_Toc31884)

[3.4.1.15 Triangle (To be worked on) 143](#_Toc1669)

[3.5. INHERITANCE (Functions, Methods, Class Hierarchies, Polymorphism, Composition, Single vs Multiple Inheritance) 143](#_Toc20698)

[3.5.1.1 OOP Fundamentals: Inheritance - why and how? 143](#_Toc15305)

[3.5.1.2 OOP Fundamentals: Inheritance - why and how? 144](#_Toc21797)

[3.5.1.3 OOP Fundamentals: Inheritance - why and how? 144](#_Toc4809)

[3.5.1.4 OOP Fundamentals: Inheritance: issubclass() 145](#_Toc13350)

[3.5.1.5 OOP Fundamentals: Inheritance: isinstance() 146](#_Toc6881)

[3.5.1.6 OOP Fundamentals: Inheritance: the is operator 147](#_Toc14638)

[3.5.1.7 How Python finds properties and methods 1/12/22 148](#_Toc3575)

[3.5.1.8 How Python finds properties and methods: continued 148](#_Toc19197)

[3.5.1.9 How Python finds properties and methods: continued 149](#_Toc17017)

[3.5.1.10 How Python finds properties and methods: continued 149](#_Toc28235)

[3.5.1.11 How Python finds properties and methods: continued 150](#_Toc6713)

[3.5.1.12 How Python finds properties and methods: continued 151](#_Toc5551)

[3.5.1.13 How Python finds properties and methods: continued 151](#_Toc15671)

[3.5.1.14 How Python finds properties and methods: continued 152](#_Toc29140)

[3.5.1.15 How to build a hierarchy of classes 153](#_Toc10128)

[3.5.1.16 How to build a hierarchy of classes: continued 154](#_Toc19468)

[3.5.1.17 How to build a hierarchy of classes: continued 1/13/22 155](#_Toc3129)

[3.5.1.18 How to build a hierarchy of classes: continued 156](#_Toc1732)

[3.5.1.19 Single inheritance vs. multiple inheritance 157](#_Toc23076)

[3.5.1.20 OOP Fundamentals: MRO 157](#_Toc31341)

[3.5.1.21 OOP Fundamentals: MRO: The diamond problem 159](#_Toc24582)

[3.5.1.22 SECTION SUMMARY 1/2 160](#_Toc23250)

[3.5.1.23 SECTION SUMMARY 2/2 162](#_Toc10969)

[3.6. THE OBJECTIVE NATURE OF PYTHON EXCEPTIONS 163](#_Toc28660)

[3.6.1.1 Exceptions once again: More about exceptions 1/14/22 163](#_Toc24004)

[3.6.1.2 Exceptions once again: More about exceptions 164](#_Toc1757)

[3.6.1.3 Exceptions once again:Exceptions are classes 165](#_Toc30242)

[3.6.1.4 Exceptions once again: Exceptions are classes 165](#_Toc25108)

[3.6.1.5 Exceptions once again: Detailed anatomy of exceptions 169](#_Toc17580)

[3.6.1.6 How to create your own exception 170](#_Toc28301)

[3.6.1.7 How to create your own exception:continued 170](#_Toc27930)

[3.6.1.8 How to create your own exception:continued 1/15/22 171](#_Toc26963)

[3.6.1.9 SECTION SUMMARY: Key takeaways 172](#_Toc1980)

[3.6.1.10 Module Completion 174](#_Toc17208)

[3.7 PE2 -- Module 3 Quiz 174](#_Toc7377)

[Module 3 Test 176](#_Toc15182)

[MODULE 4: MISCELLANEOUS 179](#_Toc12203)

[4.1. GENERATORS, ITERATORS, AND CLOSURES 179](#_Toc11778)

[4.1.1.0 Python Essentials 2 - Module 4 179](#_Toc32739)

[4.1.1.1 Generators and closures: Generators - where to find them 179](#_Toc17918)

[4.1.1.2 Generators and closures: Generators - where to find them: continued 180](#_Toc6434)

[4.1.1.3 Generators and closures: Generators - where to find them: continued 182](#_Toc25929)

[4.1.1.4 Generators and closures: The yield statement 183](#_Toc20048)

[4.1.1.5 Generators and closures: How to build your own generator 184](#_Toc4155)

[4.1.1.6 Generators and closures: More about list comprehensions 186](#_Toc25385)

[4.1.1.7 Generators and closures: More about list comprehensions 186](#_Toc12608)

[4.1.1.8 Generators and closures: More about list comprehensions 187](#_Toc32647)

[4.1.1.9 Generators and closures: The lambda function 188](#_Toc894)

[4.1.1.10 Generators and closures: How to use lambdas and what for? 189](#_Toc8672)

[4.1.1.11 Generators and closures: Lambdas and the map() function 1/16/22 190](#_Toc9177)

[4.1.1.12 Generators and closures: Lambdas and the filter() function 190](#_Toc12996)

[4.1.1.13 Generators and closures: A brief look at closures 191](#_Toc747)

[4.1.1.14 Generators and closures: A brief look at closures: continued 192](#_Toc10607)

[4.1.1.15 SECTION SUMMARY: Key takeaways 193](#_Toc2025)

[4.2. FILES (File Streams, File Processing, Diagnosing Streaming Problems) 195](#_Toc29084)

[4.2.1.1 Processing files:Accessing files from Python code 195](#_Toc29583)

[4.2.1.2 Processing files: File names 195](#_Toc1343)

[4.2.1.3 Processing files: File names: continued 196](#_Toc19335)

[4.2.1.4 Processing files: File streams 197](#_Toc215)

[4.2.1.5 Processing files: File handles 198](#_Toc6630)

[4.2.1.6 Processing files: File handles: continued 198](#_Toc28523)

[4.2.1.7 Processing files: File handles: continued 199](#_Toc27060)

[4.2.1.8 Processing files: Opening the streams: modes 200](#_Toc13181)

[4.2.1.9 Processing files: Opening the stream for the first time 201](#_Toc28832)

[4.2.1.10 Processing files: Closing streams 1/17/22 202](#_Toc23665)

[4.2.1.11 Processing files: Diagnosing stream problems: continued 203](#_Toc5954)

[4.2.1.12 Section Summary: Key takeaways 204](#_Toc29747)

[4.3. PROCESSING TEXT AND BINARY FILES 205](#_Toc23716)

[4.3.1.1 Working with real files: Processing text files 205](#_Toc28462)

[4.3.1.2 Working with real files: Processing text files: continued 1/18/22 206](#_Toc14062)

[4.3.1.3 Working with real files: Processing text files: continued 207](#_Toc30920)

[4.3.1.4 Working with real files: Processing text files: readline() 208](#_Toc25174)

[4.3.1.5 Working with real files: Processing text files: readline() 209](#_Toc26889)

[4.3.1.6 Working with real files: Processing text files: continued 1/19/22 210](#_Toc22736)

[4.3.1.7 Working with real files: Dealing with text files: write() 210](#_Toc21516)

[4.3.1.8 Working with real files: Dealing with text files: continued 211](#_Toc16405)

[4.3.1.9 Working with real files: What is a bytearray? 212](#_Toc27246)

[4.3.1.10 Working with real files: Bytearrays: continued 212](#_Toc18677)

[4.3.1.11 Working with real files: Bytearrays: continued 213](#_Toc23009)

[4.3.1.12 Working with real files: How to read bytes from a stream 214](#_Toc4616)

[4.3.1.13 Working with real files: How to read bytes from a stream: continued 214](#_Toc6014)

[4.3.1.14 Working with real files: Copying files - a simple and functional tool 215](#_Toc29002)

[4.3.1.15 LAB: Character frequency histogram 216](#_Toc2452)

[4.3.1.16 LAB: Sorted character frequency histogram 216](#_Toc25371)

[4.3.1.17 LAB: Evaluating students' results 217](#_Toc29956)

[4.3.1.18 SECTION SUMMARY: Key takeaways 1/20/22 217](#_Toc4795)

[4.4. THE OS MODULE - INTERACTING WITH THE OPERATING SYSTEM 218](#_Toc29293)

[4.4.1.1 The os module: Introduction to the os module 218](#_Toc20292)

[4.4.1.2 The os module: Getting information about the operating system 218](#_Toc4523)

[4.4.1.3 The os module: Creating directories in Python 219](#_Toc6557)

[4.4.1.4 The os module: Recursive directory creation 220](#_Toc18052)

[4.4.1.5 The os module: Where am I now? 220](#_Toc6923)

[4.4.1.6 The os module: Deleting directories in Python 221](#_Toc20335)

[4.4.1.7 The os module: The system() function 222](#_Toc4425)

[4.4.1.8 The os module: LAB 222](#_Toc4055)

[4.4.1.9 SECTION SUMMARY: Key takeaways 223](#_Toc25673)

[4.5. THE DATETIME AND TIME MODULES - WORKING WITH DATE- AND TIME-RELATED FUNCTIONS 225](#_Toc24914)

[4.5.1.1 The datetime module: Introduction to the datetime module 1/21/22 225](#_Toc31498)

[4.5.1.2 The datetime module: Getting the current local date and creating date objects 225](#_Toc15258)

[4.5.1.3 The datetime module: Creating a date object from a timestamp 226](#_Toc20528)

[4.5.1.4 The datetime module: Creating a date object using the ISO format 227](#_Toc20816)

[4.5.1.5 The datetime module: The replace() method 227](#_Toc3130)

[4.5.1.6 The datetime module: What day of the week is it? 227](#_Toc28622)

[4.5.1.7 The datetime module: Creating time objects 228](#_Toc23179)

[4.5.1.8 The datetime module: The time module 228](#_Toc8852)

[4.5.1.9 The datetime module: The ctime() function 229](#_Toc6055)

[4.5.1.10 The datetime module: The gmtime() and localtime() functions 229](#_Toc24341)

[4.5.1.11 The datetime module: The asctime() and mktime() functions 230](#_Toc10823)

[4.5.1.12 The datetime and time modules: Creating datetime objects 231](#_Toc25580)

[4.5.1.13 The datetime and time modules: Methods that return the current date and time 232](#_Toc9788)

[4.5.1.14 The datetime and time modules: Getting a timestamp 1/22/22 232](#_Toc3781)

[4.5.1.15 The datetime and time modules: Date and time formatting (part 1) 233](#_Toc3891)

[4.5.1.16 The datetime and time modules: Date and time formatting (part 2) 233](#_Toc18043)

[4.5.1.17 The datetime and time modules: The strftime() function in the time module 234](#_Toc733)

[4.5.1.18 The datetime and time modules: The strptime() method 234](#_Toc17211)

[4.5.1.19 The datetime and time modules: Date and time operations 235](#_Toc3332)

[4.5.1.20 The datetime and time modules: Creating timedelta objects 235](#_Toc3246)

[4.5.1.21 The datetime and time modules: Creating timedelta objects: continued 236](#_Toc15762)

[4.5.1.22 The datetime and time modules: LAB 237](#_Toc1650)

[4.5.1.23 SECTION SUMMARY: Key takeaways 237](#_Toc6044)

[4.6. The Calendar Module - Working With Calendar - Related Functions 239](#_Toc25969)

[4.6.1.1 The calendar module: Introduction to the calendar module 239](#_Toc31956)

[4.6.1.2 The calendar module: Your first calendar 240](#_Toc25740)

[4.6.1.3 The calendar module: Calendar for a specific month 241](#_Toc1954)

[4.6.1.4 The calendar module: The setfirstweekday() function 241](#_Toc7316)

[4.6.1.5 The calendar module: The weekday() function 241](#_Toc31823)

[4.6.1.6 The calendar module: The weekheader() function 242](#_Toc30167)

[4.6.1.7 The calendar module: How do we check if a year is a leap year? 242](#_Toc8645)

[4.6.1.8 The calendar module: Classes for creating calendars 242](#_Toc27738)

[4.6.1.9 The calendar module: Creating a Calendar object 1/23/22 243](#_Toc6572)

[4.6.1.10 The calendar module: The itermonthdates() method 243](#_Toc16889)

[4.6.1.11 The calendar module: Other methods that return iterators 244](#_Toc29178)

[4.6.1.12 The calendar module: The monthdays2calendar() method 244](#_Toc31277)

[4.6.1.13 LAB: the calendar module 245](#_Toc8220)

[4.6.1.14 SECTION SUMMARY: Key takeaways 245](#_Toc7463)

[4.6.1.15 Module Completion: 247](#_Toc6701)

[4.7 PE2 -- Module 4 Quiz 247](#_Toc17560)

[Module 4 Test 249](#_Toc10004)

# Module 1: Modules, Packages and PIP 12/26/21

In this module, you will learn about:

* importing and using Python modules;
* using some of the most useful Python standard library modules;
* constructing and using Python packages;
* PIP (Python Installation Package) and how to use it to install and uninstall ready-to-use packages from PyPI.

# 

# **1.1 INTRODUCTION TO MODULES IN PYTHON**

## 1.1.1.1 Modules: **What is a module?**

Computer code has a tendency to grow. We can say that code that doesn't grow is probably completely unusable or abandoned. A real, wanted, and widely used code develops continuously, as both users' demands and users' expectations develop in their own rhythms.

A code which is not able to respond to users' needs will be forgotten quickly, and instantly replaced with a new, better, and more flexible code. Be prepared for this, and never think that any of your programs is eventually completed. The completion is a transition state and usually passes quickly, after the first bug report. Python itself is a good example how the rule acts. Growing code is in fact a growing problem. A larger code always means tougher maintenance. Searching for bugs is always easier where the code is smaller (just as finding a mechanical breakage is simpler when the machinery is simpler and smaller).

Moreover, when the code being created is expected to be really big (you can use a total number of source lines as a useful, but not very accurate, measure of a code's size) you may want (or rather, you will be forced) to divide it into many parts, implemented in parallel by a few, a dozen, several dozen, or even several hundred individual developers.

Of course, this cannot be done using one large source file, which is edited by all programmers at the same time. This will surely lead to a spectacular disaster. If you want such a software project to be completed successfully, you have to have the means allowing you to:

* divide all the tasks among the developers;
* join all the created parts into one working whole.

For example, a certain project can be divided into two main parts:

* the user interface (the part that communicates with the user using widgets and a graphical screen)
* the logic (the part processing data and producing results)

Each of these parts can be (most likely) divided into smaller ones, and so on. Such a process is often called **decomposition**. For example, if you were asked to arrange a wedding, you wouldn't do everything yourself - you would find a number of professionals and split the task between them all. How do you divide a piece of software into separate but cooperating parts? This is the question. **Modules** are the answer.

## 1.1.1.2 Using Modules: **How to make use of a module?**

The Python Tutorial (https://docs.python.org/3/tutorial/modules.html) defines a module as a file containing Python definitions and statements, which can be later imported and used when necessary. The handling of modules consists of two different issues:

* the first (probably the most common) happens when you want to use an already existing module, written by someone else, or created by yourself during your work on some complex project - in this case you are the module's **user**;
* the second occurs when you want to create a brand new module, either for your own use, or to make other programmers' lives easier - you are the module's **supplier**.

Let's discuss them separately.

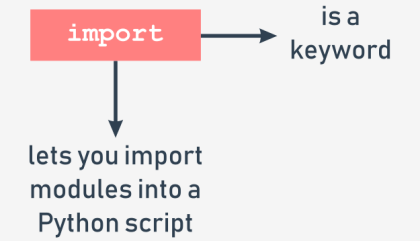
First of all, a module is identified by its **name**. If you want to use any module, you need to know the name. A (rather large) number of modules is delivered together with Python itself. You can think of them as a kind of "Python extra equipment".

All these modules, along with the built-in functions, form the Python standard library - a special sort of library where modules play the roles of books (we can even say that folders play the roles of shelves). If you want to take a look at the full list of all "volumes" collected in that library, you can find it here: https://docs.python.org/3/library/index.html.

Each module consists of entities (like a book consists of chapters). These entities can be functions, variables, constants, classes, and objects. If you know how to access a particular module, you can make use of any of the entities it stores.

Let's start the discussion with one of the most frequently used modules, named *math*. Its name speaks for itself - the module contains a rich collection of entities (not only functions) which enable a programmer to effectively implement calculations demanding the use of mathematical functions, like sin() or log().

## 1.1.1.3 Using Modules: **Importing a module**

To make a module usable, you must **import** it (think of it like of taking a book off the shelf). Importing a module is done by an instruction named *import*. Note: *import* is also a keyword (with all the consequences of this fact). Let's assume that you want to use two entities provided by the math module:

* a symbol (constant) representing a precise (as precise as possible using double floating-point arithmetic) value of π (although using a Greek letter to name a variable is fully possible in Python, the symbol is named **pi** - it's a more convenient solution, especially for those who neither has nor is going to use a Greek keyboard)
* a function named *sin*() (the computer equivalent of the math sine function)

Both these entities are available through the math module, but the way in which you can use them strongly depends on how the import has been done. The simplest way to import a particular module is to use the import instruction:

|  |
| --- |
| **import** math |

The clause contains:

* the import keyword;
* the **name of the module** which is subject to import.

The instruction may be located anywhere in your code, but it must be placed **before the first use of any of the module's entities.** If you want to (or have to) import more than one module, you can do it by repeating the import clause (preferred):

|  |
| --- |
| **import** math  **import** sys |

or by listing the modules after the import keyword, like here:

|  |
| --- |
| **import** math**,** sys |

The instruction imports two modules, first the one named *math* and then the second named *sys*. The modules' list may be arbitrarily long.

## 1.1.1.4 Using Modules: **Importing a module continued**

To continue, you need to become familiar with an important term: **namespace**. Don't worry, we won't go into detail. This explanation will be as short as possible.

A **namespace** is a space (understood in a non-physical context) in which some names exist and the names don't conflict with each other (i.e., there are not two different objects of the same name). We can say that each social group is a namespace - the group tends to name each of its members in a unique way (e.g., parents won't give their children the same first names). This uniqueness may be achieved in many ways, e.g., by using nicknames along with the first names (it will work inside a small group like a class in a school) or by assigning special identifiers to all members of the group (the US Social Security Number is a good example of such practice). **Inside a certain namespace, each name must remain unique**. This may mean that some names may disappear when any other entity of an already known name enters the namespace. We'll show you how it works and how to control it, but first, let's return to imports.

If the module of a specified name **exists and is accessible** (a module is in fact a Python source file), Python imports its contents, i.e., **all the names defined in the module become known**, but they don't enter your code's namespace. This means that you can have your own entities named sin or pi and they won't be affected by the import in any way.

At this point, you may be wondering how to access the pi coming from the math module. To do this, you have to qualify the pi with the name of its original module.

## 1.1.1.5 **Importing a module | math**

**Importing a module: continued**

Look at the snippet below, this is the way in which you qualify the names of pi and sin with the name of its originating module:

|  |
| --- |
| math.pi  math.sin |

It's simple, you put:

* the **name of the module** (e.g., math)
* a **dot** (i.e., .)
* the **name of the entity** (e.g., pi)

Such a form clearly indicates the namespace in which the name exists.

Note: using this qualification is **compulsory(必须的)** if a module has been imported by the *import* module instruction. It doesn't matter if any of the names from your code and from the module's namespace are in conflict or not.

This first example won't be very advanced - we just want to print the value of sin(½π). Look at the code in the editor. This is how we test it.

|  |
| --- |
| **import** math  **print(**math**.**sin**(**math**.**pi**/**2**))** |

The code outputs the expected value: 1.0.

Note: removing any of the two qualifications will make the code erroneous. There is no other way to enter math's namespace if you did the following:

|  |
| --- |
| **import** math |

## 

## 1.1.1.6 **Importing a module | math**

**Importing a module: continued**

Now we're going to show you how the two namespaces (yours and the module's one) can coexist. Take a look at the example in the editor window.

|  |
| --- |
| **import** math  **def** sin**(**x**):**  **if** 2 **\*** x **==** pi**:**  **return** 0.99999999  **else:**  **return** **None**  pi **=** 3.14  **print(**sin**(**pi**/**2**))**  **print(**math**.**sin**(**math**.**pi**/**2**))** |

We've defined our own pi and sin here. Run the program. The code’s output:

|  |
| --- |
| 0.99999999  1.0 |

As you can see, the entities don't affect each other.

## 1.1.1.7 **Importing a module | math**

**Importing a module: continued**

In the second method, the import's syntax precisely points out which module's entity (or entities) are acceptable in the code:

|  |
| --- |
| **from** math **import** pi |

The instruction consists of the following elements:

* the *from* keyword;
* the **name of the module** to be (selectively) imported;
* the *import* keyword;
* the **name or list of names of the entity/entities** which are being imported into the namespace.

The instruction has this effect:

* the listed entities (and only those ones) are **imported from the indicated module**;
* the names of the imported entities are **accessible without qualification**.

Note: no other entities are imported. Moreover, you cannot import additional entities using a qualification - a line like this one:

|  |
| --- |
| **print(**math**.**e**)** |

will cause an error (e is Euler's number: 2.71828...)

Let's rewrite the previous script to incorporate the new technique. Here it is:

|  |
| --- |
| **from** math **import** sin**,** pi  **print(**sin**(**pi**/**2**))** |

The output should be the same as previously, as in fact we've used the same entities as before: 1.0. Copy the code, paste it in the editor, and run the program.

Does the code look simpler? Maybe, but the look is not the only effect of this kind of import. Let's show you that.

## 1.1.1.8 **Importing a module | math**

**Importing a module: continued**

Look at the code in the editor.

|  |
| --- |
| **from** math **import** sin**,** pi  **print(**sin**(**pi **/** 2**))**  pi **=** 3.14  **def** sin**(**x**):**  **if** 2 **\*** x **==** pi**:**  **return** 0.99999999  **else:**  **return** **None**  **print(**sin**(**pi **/** 2**))** |

Analyze it carefully:

* line 1: carry out the selective import;
* line 2: make use of the imported entities and get the expected result (1.0)
* lines 3 through 9: redefine the meaning of pi and sin - in effect, they supersede the original (imported) definitions within the code's namespace;
* line 10: get 0.99999999, which confirms our conclusions.

Let's do another test. Look at the code below:

|  |
| --- |
| pi **=** 3.14  **def** sin**(**x**):**  **if** 2 **\*** x **==** pi**:**  **return** 0.99999999  **else:**  **return** **None**  **print(**sin**(**pi **/** 2**))**  **from** math **import** sin**,** pi  **print(**sin**(**pi **/** 2**))** |

Here, we've reversed the sequence of the code's operations:

* lines 1 through 6: define our own pi and sin;
* line 7: make use of them (0.99999999 appears on the screen)
* line 8: carry out the import - the imported symbols **supersede their previous definitions within the namespace**;
* line 9 get 1.0 as a result.

## **1.1.1.9 Importing a module | \* and as**

**Importing a module: \***

In the third method, the import's syntax is a more aggressive form of the previously presented one:

|  |
| --- |
| **from** module **import** **\*** |

As you can see, the name of an entity (or the list of entities' names) is replaced with a single asterisk (\*). Such an instruction **imports all entities from the indicated module**. Is it convenient? Yes, it is, as it relieves you of the duty of enumerating all the names you need. Is it unsafe? Yes, it is - unless you know all the names provided by the module, **you may not be able to avoid name conflicts**. Treat this as a temporary solution, and try not to use it in regular code.

**Importing a module: the as keyword**

If you use the import module variant and you don't like a particular module's name (e.g., it's the same as one of your already defined entities, so qualification becomes troublesome) you can give it any name you like - this is called **aliasing**.

Aliasing causes the module to be identified under a different name than the original. This may shorten the qualified names, too. Creating an alias is done together with importing the module, and demands the following form of the import instruction:

|  |
| --- |
| **import** module **as** alias |

The "module" identifies the original module's name while the "alias" is the name you wish to use instead of the original. Note: *as* is a keyword.

## **1.1.1.10 Importing a module | aliasing**

**Importing a module: continued**

If you need to change the word *math*, you can introduce your own name:

|  |
| --- |
| **import** math **as** m  **print(**m**.**sin**(**m**.**pi**/**2**))** |

Note: after successful execution of an aliased import, the **original module name becomes inaccessible** and must not be used.

In turn, when you use the from module import name variant and you need to change the entity's name, you make an alias for the entity. This will cause the name to be replaced by the alias you choose.This is how it can be done:

|  |
| --- |
| **from** module **import** name **as** alias |

As previously, the original (unaliased) name becomes inaccessible. The phrase *name as alias* can be repeated - use commas to separate the multiplied phrases, like this:

|  |
| --- |
| **from** module **import** n **as** a**,** m **as** b**,** o **as** c |

The example may look a bit weird, but it works:

|  |
| --- |
| **from** math **import** pi **as** PI**,** sin **as** sine  **print(**sine**(**PI**/**2**))** |

Now you're familiar with the basics of using modules. Let us show you some modules and some of their useful entities.

## **1.1.1.11 SECTION SUMMARY: Key takeaways**

1. If you want to import a module as a whole, you can do it using the import module\_name statement. You are allowed to import more than one module at once using a comma-separated list. For example:

|  |
| --- |
| **import** mod1  **import** mod2**,** mod3**,** mod4 |

although the latter form is not recommended due to stylistic reasons, and it's better, prettier to express the same intention in more a verbose and explicit form, such as:

|  |
| --- |
| **import** mod2  **import** mod3  **import** mod4 |

1. If a module is imported in the above manner and you want to access any of its entities, you need to prefix the entity's name using **dot notation**. For example:

|  |
| --- |
| **import** my\_module  result **=** my\_module**.**my\_function**(**my\_module**.**my\_data**)** |

The snippet makes use of two entities coming from the my\_module module: a function named my\_function() and a variable named my\_data. Both names **must be prefixed by** my\_module. None of the imported entity names conflicts with the identical names existing in your code's namespace.

1. You are allowed not only to import a module as a whole, but to import only individual entities. In this case, the imported entities must not be prefixed when used. For example:

|  |
| --- |
| **from** module **import** my\_function**,** my\_data  result **=** my\_function**(**my\_data**)** |

The above way - despite its attractiveness - is not recommended because of the danger of causing conflicts with names derived from importing the code's namespace.

4. The most general form of the above statement allows you to import all entities offered by a module:

|  |
| --- |
| **from** module **import** \*  result **=** my\_function**(**my\_data**)** |

**Note**: this import's variant is not recommended due to the same reasons as previously (the threat of a naming conflict is even more dangerous here).

1. You can change the name of the imported entity "on the fly" by using the as phrase of the import. For example:

|  |
| --- |
| **from** module **import** my\_function **as** fun**,** my\_data **as** dat  result **=** fun**(**dat**)** |

**Exercise 1**

You want to invoke the function make\_money() contained in the module named mint. Your code begins with the following line: import mint

What is the proper form of the function's invocation?

Check: mint.make\_money()

**Exercise 2**

You want to invoke the function make\_money() contained in the module named mint. Your code begins with the following line:

from mint import make\_money

What is the proper form of the function's invocation?

Check: make\_money()

**Exercise 3**

You've written a function named make\_money on your own. You need to import a function of the same name from the mint module and don't want to rename any of your previously defined names. Which variant of the import statement may help you with the issue?

Check: # sample solution

from mint import make\_money as make\_more\_money

**Exercise 4**

What form of the make\_money function invocation is valid if your code starts with the following line?

from mint import \*

Check: make\_money()

# 1.2. SELECTED PYTHON MODULES (MATH, RANDOM, PLATFORM)

## **1.2.1.1 Useful Modules: Working with standard modules**

Before we start going through some standard Python modules, we want to introduce the *dir*() function to you. It has nothing to do with the *dir* command you know from Windows and Unix consoles, as *dir*() doesn't show the contents of a disk directory/folder, but there is no denying that it does something really similar - it is able to reveal all the names provided through a particular module.

There is one condition: the module has to have been previously imported as a whole (i.e., using the import module instruction - from module is not enough).

The function returns an **alphabetically sorted** list containing all entities' names available in the module identified by a name passed to the function as an argument:

|  |
| --- |
| **dir(**module**)** |

Note: if the module's name has been aliased, you must use the alias, not the original name. Using the function inside a regular script doesn't make much sense, but it is still possible. For example, you can run the following code to print the names of all entities within the math module:

|  |
| --- |
| **import** math  **for** name **in** **dir(**math**):**  **print(**name**,** end**=**"\t"**)** |

The example code should produce the following output:

|  |
| --- |
| \_\_doc\_\_ \_\_loader\_\_ \_\_name\_\_ \_\_package\_\_ \_\_spec\_\_ acos acosh asin asinh atan atan2 atanh ceil copysign cos cosh degrees e erf erfc exp expm1 fabs factorial floor fmod frexp fsum gamma hypot isfinite isinf isnan ldexp lgamma log log10 log1p log2 modf pi pow radians sin sinh sqrt tan tanh trunc |

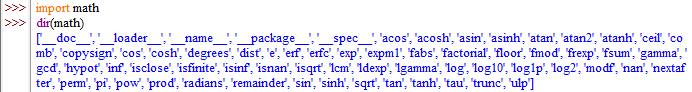
Have you noticed these strange names beginning with \_\_ at the top of the list? We'll tell you more about them when we talk about the issues related to writing your own modules.Some of the names might bring back memories from math lessons, and you probably won't have any problems guessing their meanings.

Using the dir() function inside a code may not seem very useful - usually you want to know a particular module's contents before you write and run the code.

Fortunately, you can execute the function directly in the Python console (IDLE), without needing to write and run a separate script.This is how it can be done:

|  |
| --- |
| **import** math  **dir(**math**)** |

You should see something similar to this:



## **1.2.1.2 Useful modules | math: Selected functions from the math module**

Let's start with a quick preview of some of the functions provided by the *math* module.We've chosen them arbitrarily, but that doesn't mean that the functions we haven't mentioned here are any less significant. Dive into the modules' depths yourself - we don't have the space or the time to talk about everything in detail here.

The first group of the math's functions are connected with **trigonometry**:

* sin(x) → the sine of x;
* cos(x) → the cosine of x;
* tan(x) → the tangent of x.

All these functions take one argument (an angle measurement expressed in radians), return the appropriate result (be careful with tan() - not all arguments are accepted).

Of course, there are also their inversed versions:

* asin(x) → the arcsine of x;
* acos(x) → the arccosine of x;
* atan(x) → the arctangent of x.

These functions take one argument (mind the domains) and return a measure of an angle in radians. To effectively operate on angle measurements, the math module provides you with the following entities:

* pi → a constant with a value that is an approximation of π;
* radians(x) → a function that converts x from degrees to radians;
* degrees(x) → acting in the other direction (from radians to degrees)

Now look at the code in the editor. The example program isn't very sophisticated, but can you predict its results?

from math import pi, radians, degrees, sin, cos, tan, asin

ad = 90

ar = radians(ad)

ad = degrees(ar)

print(ad == 90.)

print(ar == pi / 2.)

print(sin(ar) / cos(ar) == tan(ar))

print(asin(sin(ar)) == ar)

Output:

True

True

True

True

Apart from the circular functions (listed above) the math module also contains a set of their hyperbolic analogues:

* sinh(x) → the hyperbolic sine;
* cosh(x) → the hyperbolic cosine;
* tanh(x) → the hyperbolic tangent;
* asinh(x) → the hyperbolic arcsine;
* acosh(x) → the hyperbolic arccosine;
* atanh(x) → the hyperbolic arctangent.

## 

## **1.2.1.3 Useful modules | math: Selected functions from the math module: continued**

Another group of the math's functions is formed by functions which are connected with **exponentiation**:

* e → a constant with a value that is an approximation of Euler's number (e)
* exp(x) → finding the value of e^x;
* log(x) → the natural logarithm of x
* log(x, b) → the logarithm of x to base b
* log10(x) → the decimal logarithm of x (more precise than log(x, 10))
* log2(x) → the binary logarithm of x (more precise than log(x, 2))

Note: the pow() function:

* pow(x, y) → finding the value of xy (mind the domains)
* This is a built-in function, and doesn't have to be imported.

Look at the code in the editor. Can you predict its output?

from math import e, exp, log

print(pow(e, 1) == exp(log(e)))

print(pow(2, 2) == exp(2 \* log(2)))

print(log(e, e) == exp(0))

Output

False

True

True

## 

## **1.2.1.4 Useful modules | math: Selected functions from the math module: continued**

The last group consists of some general-purpose functions like:

* ceil(x) → the ceiling of x (the smallest integer greater than or equal to x)
* floor(x) → the floor of x (the largest integer less than or equal to x)
* trunc(x) → the value of x truncated to an integer (be careful - it's not an equivalent either of ceil or floor)
* factorial(x) → returns x! (x has to be an integral and not a negative)
* hypot(x, y) → returns the length of the hypotenuse of a right-angle triangle with the leg lengths equal to x and y (the same as sqrt(pow(x, 2) + pow(y, 2)) but more precise)

Look at the code in the editor. Analyze the program carefully.

from math import ceil, floor, trunc

x = 1.4

y = 2.6

print(floor(x), floor(y))

print(floor(-x), floor(-y))

print(ceil(x), ceil(y))

print(ceil(-x), ceil(-y))

print(trunc(x), trunc(y))

print(trunc(-x), trunc(-y))

Output:

1 2

-2 -3

2 3

-1 -2

1 2

-1 -2

It demonstrates the fundamental differences between ceil(), floor() and trunc().

## **1.2.1.5 Useful modules | random: Is there real randomness in computers?**

Another module worth mentioning is the one named *random*. It delivers some mechanisms allowing you to operate with **pseudorandom numbers.**

Note the prefix **pseudo** - the numbers generated by the modules may look random in the sense that you cannot predict their subsequent values, but don't forget that they all are calculated using very refined algorithms.

The algorithms aren't random - they are deterministic and predictable. Only those physical processes which run completely out of our control (like the intensity of cosmic radiation) may be used as a source of actual random data. Data produced by deterministic computers cannot be random in any way.

A random number generator takes a value called a **seed**, treats it as an input value, calculates a "random" number based on it (the method depends on a chosen algorithm) and produces a **new seed value**.

The length of a cycle in which all seed values are unique may be very long, but it isn't infinite - sooner or later the seed values will start repeating, and the generating values will repeat, too. This is normal. It's a feature, not a mistake, or a bug. The initial seed value, set during the program start, determines the order in which the generated values will appear.

The random factor of the process may be **augmented by setting the seed with a number taken from the current time** - this may ensure that each program launch will start from a different seed value (ergo, it will use different random numbers). Fortunately, such an initialization is done by Python during module import.

## **1.2.1.6 Useful modules | random**

**Selected functions from the random module: The random function**

The most general function named random() (not to be confused with the module's name) produces a float number x coming from the range (0.0, 1.0) - in other words: (0.0 <= x < 1.0).

The example program below will produce five pseudorandom values - as their values are determined by the current (rather unpredictable) seed value, you can't guess them:

|  |
| --- |
| **from** random **import** random  **for** i **in** **range(**5**):**  **print(**random**())** |

Run the program. This is what we've got:

0.9535768927411208

0.5312710096244534

0.8737691983477731

0.5896799172452125

0.02116716297022092

**The seed function**

The *seed*() function is able to directly **set the generator's seed**. We'll show 2 variants:

* seed() - sets the seed with the current time;
* seed(int\_value) - sets the seed with the integer value int\_value.

We've modified the previous program - in effect, we've removed any trace of randomness from the code:

|  |
| --- |
| **from** random **import** random**,** seed  seed**(**0**)**  **for** i **in** **range(**5**):**  **print(**random**())** |

Due to the fact that the seed is always set with the same value, the sequence of generated values always looks the same. Run the program. This is what we've got:

0.844421851525

0.75795440294

0.420571580831

0.258916750293

0.511274721369

Note: your values may be slightly different than ours if your system uses more/less precise floating-point arithmetic, but the difference will be seen quite far from the decimal point.

## **1.2.1.7 Useful modules | random**

**Selected functions from the random module: continued**

**The *randrange* and *randint* functions：**If you want integer random values, one of the following functions would fit better:

* randrange(end)
* randrange(beg, end)
* randrange(beg, end, step)
* randint(left, right)

The first three invocations will generate an integer taken (pseudorandomly) from the range (respectively):

* range(end)
* range(beg, end)
* range(beg, end, step)

Note the implicit **right-sided exclusion**!

The last function is an equivalent of randrange(left, right+1) - it generates the integer value i, which falls in the range [left, right] (no exclusion on the right side). Look at the code in the editor. This sample program will consequently output a line consisting of three zeros and either a zero or one at the fourth place.

|  |
| --- |
| **from** random **import** randrange**,** randint  **print(**randrange**(**1**),** end**=**' '**)**  **print(**randrange**(**0**,** 1**),** end**=**' '**)**  **print(**randrange**(**0**,** 1**,** 1**),** end**=**' '**)**  **print(**randint**(**0**,** 1**))** |

Output: 0 0 0 0

## **1.2.1.8 Useful modules | random**

**Selected functions from the random module: continued**

The previous functions have one important disadvantage - they may produce repeating values even if the number of subsequent invocations is not greater than the width of the specified range. Look at the code below - the program very likely outputs a set of numbers in which some elements are not unique:

|  |
| --- |
| **from** random **import** randint  **for** i **in** **range(**10**):**  **print(**randint**(**1**,** 10**),** end**=**','**)** |

This is what we got in one of the launches:9,4,5,4,5,8,9,4,8,4,

**The *choice* and *sample* functions**

As you can see, this is not a good tool for generating numbers in a lottery. Fortunately, there is a better solution than writing your own code to check the uniqueness of the "drawn" numbers. It's a function named in a very suggestive way - *choice*:

* choice(sequence)
* sample(sequence, elements\_to\_choose)

The first variant chooses a "random" element from the input sequence and returns it. The second one builds a list (a sample) consisting of the *elements\_to\_choose* element "drawn" from the input sequence.

In other words, the function chooses some of the input elements, returning a list with the choice. The elements in the sample are placed in random order. Note: the *elements\_to\_choose* must not be greater than the length of the input sequence. Look at the code below:

|  |
| --- |
| **from** random **import** choice**,** sample  my\_list **=** **[**1**,** 2**,** 3**,** 4**,** 5**,** 6**,** 7**,** 8**,** 9**,** 10**]**  **print(**choice**(**my\_list**))**  **print(**sample**(**my\_list**,** 5**))**  **print(**sample**(**my\_list**,** 10**))** |

Again, the output of the program is not predictable. Our results looked like this:

|  |
| --- |
| 4  [3, 1, 8, 9, 10]  [10, 8, 5, 1, 6, 4, 3, 9, 7, 2] |

## 

## **1.2.1.9 Useful modules | platform 12/27/21**

**How to know where you are?**

Sometimes, it may be necessary to find out information unrelated to Python. For example, you may need to know the location of your program within the greater environment of the computer. Imagine your program's environment as a pyramid consisting of a number of layers or platforms.

The layers are:

* your (running) code is located at the top of it;
* Python (more precisely - its runtime environment) lies directly below it;
* the next layer of the pyramid is filled with the OS (operating system) - Python's environment provides some of its functionalities using the operating system's services; Python, although very powerful, isn't omnipotent - it's forced to use many helpers if it's going to process files or communicate with physical devices;
* the bottom-most layer is hardware - the processor (or processors), network interfaces, human interface devices (mice, keyboards, etc.) and all other machinery needed to make the computer run; the OS knows how to drive it, and uses lots of tricks to conduct all parts in a consistent rhythm.

This means than some of your (or rather your program's) actions have to travel a long way to be successfully performed - imagine that:

* **your code** wants to create a file, so it invokes one of Python's functions;
* **Python** accepts the order, rearranges it to meet local OS requirements (it's like putting the stamp "approved" on your request) and sends it down (this may remind you of a chain of command)
* the **OS** checks if the request is reasonable and valid (e.g., whether the file name conforms to some syntax rules) and tries to create the file; such an operation, seemingly very simple, isn't atomic - it consists of many minor steps taken by...
* the **hardware**, which is responsible for activating storage devices (hard disk, solid state devices, etc.) to satisfy the OS's needs.

Usually, you're not aware of all that fuss - you want the file to be created and that's that. But sometimes you want to know more - for example, the name of the OS which hosts Python, and some characteristics describing the hardware that hosts the OS.

There is a module providing some means to allow you to know where you are and what components work for you. The module is named **platform**. We'll show you some of the functions it provides to you.

## **1.2.1.10 Useful modules | platform: platform function**

**Selected functions from the platform module**

**The platform function**

The *platform* module lets you access the underlying platform's data, i.e., hardware, operating system, and interpreter version information. There is a function that can show you all the underlying layers in one glance, named *platform*, too. It just returns a string describing the environment; thus, its output is rather addressed to humans than to automated processing (you'll see it soon). This is how to invoke it:

|  |
| --- |
| platform**(**aliased **=** **False,** terse **=** **False)** |

And now:

* *aliased* → when set to *True* (or any non-zero value) it may cause the function to present the alternative underlying layer names instead of the common ones;
* *terse* → when set to *True* (or any non-zero value) it may convince the function to present a briefer form of the result (if possible)

We ran our sample program using three different platforms - this is what we got:

**Intel x86 + Windows ® Vista (32 bit):**

Windows-Vista-6.0.6002-SP2

Windows-Vista-6.0.6002-SP2

Windows-Vista

**Intel x86 + Gentoo Linux (64 bit):**

Linux-3.18.62-g6-x86\_64-Intel-R-\_Core-TM-\_i3-2330M\_CPU\_@\_2.20GHz

Linux-3.18.62-g6-x86\_64-Intel-R-\_Core-TM-\_i3-2330M\_CPU\_@\_2.20GHz

Linux-3.18.62-g6-x86\_64-Intel-R-\_Core-TM-\_i3-2330M\_CPU\_@\_2.20GHz

You can also run the sample program in IDLE on your local machine to check what output you will have.

from platform import platform

print(platform())

print(platform(1))

print(platform(0, 1))

Output：

Windows-10-10.0.19041-SP0

Windows-10-10.0.19041-SP0

Windows-10

## 

## **1.2.1.11 Useful modules | platform: machine function**

**Selected functions from the platform module: continued**

**The machine function**

Sometimes, you may just want to know the generic name of the processor which runs your OS together with Python and your code - a function named machine() will tell you that. As previously, the function returns a string.

Again, we ran the sample program on three different platforms:

Intel x86 + Windows ® Vista (32 bit): x86

Intel x86 + Gentoo Linux (64 bit): x86\_64

Raspberry PI2 + Raspbian Linux (32 bit): armv7l

|  |
| --- |
| **from** platform **import** machine  **print(**machine**())** |

## 

## **1.2.1.12 Useful modules | platform: processor function**

**Selected functions from the platform module: continued**

**The processor function**

The *processor*() function returns a string filled with the real processor name (if possible).Once again, we ran the sample program on three different platforms:

Intel x86 + Windows ® Vista (32 bit):x86

Intel x86 + Gentoo Linux (64 bit):Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz

Raspberry PI2 + Raspbian Linux (32 bit):armv7l

Test this on your local machine.

|  |
| --- |
| **from** platform **import** processor  **print(**processor**())** |

## 

## **1.2.1.13 Useful modules | platform: system function**

**Selected functions from the platform module: continued**

**The system function**

A function named *system*() returns the generic OS name as a string. Our example platforms presented themselves like this:

Intel x86 + Windows ® Vista (32 bit): Windows

Intel x86 + Gentoo Linux (64 bit):Linux

Raspberry PI2 + Raspbian Linux (32 bit):Linux

|  |
| --- |
| **from** platform **import** system  **print(**system**())** |

## 

## **1.2.1.14 Useful modules | platform: version function**

**Selected functions from the platform module: continued**

**The version function**

The OS version is provided as a string by the version() function.Run the code and check its output. This is what we got:

Intel x86 + Windows ® Vista (32 bit):6.0.6002

Intel x86 + Gentoo Linux (64 bit):#1 SMP PREEMPT Fri Jul 21 22:44:37 CEST 2017

Raspberry PI2 + Raspbian Linux (32 bit):#1 SMP Debian 4.4.6-1+rpi14 (2016-05-05)

|  |
| --- |
| **from** platform **import** version  **print(**version**())** |

## 

## **1.2.1.15 Useful modules | platform: 2 more functions**

**Selected functions from the platform module: continued**

**The python\_implementation and the python\_version\_tuple functions**

If you need to know what version of Python is running your code, you can check it using a number of dedicated functions - here are two of them:

* *python\_implementation*() → returns a string denoting the Python implementation (expect CPython, unless you decide to use any non-canonical Python branch)
* *python\_version\_tuple*() → returns a three-element tuple filled with:
  + the **major** part of Python's version;
  + the **minor** part;
  + the **patch** level number.

Our example program produced the following output:

CPython

3

7

7

It's very likely that your version of Python will be different.

|  |
| --- |
| **from** platform **import** python\_implementation**,** python\_version\_tuple  **print(**python\_implementation**())**  **for** atr **in** python\_version\_tuple**():**  **print(**atr**)** |

## **1.2.1.16 Useful modules | Python Module Index**

We have only covered the basics of Python modules here. Python's modules make up their own universe, in which Python itself is only a galaxy, and we would venture to say that exploring the depths of these modules can take significantly more time than getting acquainted with "pure" Python.

Moreover, the Python community all over the world creates and maintains hundreds of additional modules used in very niche applications like genetics, psychology, or even astrology. These modules aren't (and won't be) distributed along with Python, or through official channels, which makes the Python universe broader - almost infinite. You can read about all standard Python modules here: https://docs.python.org/3/py-modindex.html.

Don't worry - you won't need all these modules. Many of them are very specific. All you need to do is find the modules you want, and teach yourself how to use them. It's easy. In the next section we'll take a look at something else. We're going to show you how to write your own module.

## **1.2.1.17 SECTION SUMMARY: Key takeaways**

1. A function named dir() can show you a list of the entities contained inside an imported module. For example:

|  |
| --- |
| **import** os  **dir(**os**)** |

prints out the list of all the os module's facilities you can use in your code.

2. The *math* module couples more than 50 symbols (functions and constants) that perform mathematical operations (like sine(), pow(), factorial()) or providing important values (like π and the Euler symbol e).

3. The *random* module groups more than 60 entities designed to help you use pseudo-random numbers. Don't forget the prefix "random", as there is no such thing as a real random number when it comes to generating them using the computer's algorithms.

4. The *platform* module contains about 70 functions which let you dive into the underlaying layers of the OS and hardware. Using them allows you to get to know more about the environment in which your code is executed.

5. **Python Module Index** (https://docs.python.org/3/py-modindex.html is a community-driven directory of modules available in the Python universe. If you want to find a module fitting your needs, start your search there.

**Exercise 1:** What is the expected value of the result variable after the following code is executed?

|  |
| --- |
| **import** math  result **=** math**.**e **==** math**.**exp**(**1**)** |

Check: True

**Exercise 2:** (Complete the sentence) Setting the generator's seed with the same value each time your program is run guarantees that...

Check: the pseudo-random values emitted from the *random* module will be exactly the same.

**Exercise 3:** Which of the platform module's functions will you use to determine the name of the CPU running inside your computer?

Check: The processor() function

**Exercise 4:** What is the expected output of the following snippet?

import platform

print(len(platform.python\_version\_tuple()))

Check：3

# 1.3. MODULES AND PACKAGES

## **1.3.1.1 Modules and Packages: What is a package?**

Writing your own modules doesn't differ much from writing ordinary scripts. There are some specific aspects you must be aware of, but it definitely isn't rocket science. You'll see this soon enough. Let's summarize some important issues:

* a **module is a kind of container filled with functions** - you can pack as many functions as you want into one module and distribute it across the world;
* of course, it's generally a good idea not to mix functions with different application areas within one module (just like in a library - nobody expects scientific works to be put among comic books), so group your functions carefully and name the module containing them in a clear and intuitive way (e.g., don't give the name arcade\_games to a module containing functions intended to partition and format hard disks)
* making many modules may cause a little mess - sooner or later you'll want to **group your modules** exactly in the same way as you've previously grouped functions - is there a more general container than a module?
* yes, there is - it's a **package**; in the world of modules, a package plays a similar role to a folder/directory in the world of files.

## 

## **1.3.1.2 Modules and Packages: Your first module: step 1-2**

In this section you're going to be working locally on your machine. Let's start from scratch. Create an empty file, just like this:

You will need two files to repeat these experiments. The first of them will be the module itself. It's empty now. Don't worry, you will fill it with actual code soon.

We've named the file module.py. Not very creative, but simple and clear.

**Your first module: step 2**

The second file contains the code using the new module. Its name is main.py. Its content is very brief so far:

Creating a main.py file containing the import module instruction

import module

Note: **both files have to be located in the same folder**. We strongly encourage you to create an empty, new folder for both files. Some things will be easier then.

Launch IDLE (or any other IDE you prefer) and run the main.py file. What do you see?You should see nothing. This means that Python has successfully imported the contents of the module.py file.

It doesn't matter that the module is empty for now. The very first step has been done, but before you take the next step, we want you to take a look into the folder in which both files exist. Do you notice something interesting?

A new subfolder has appeared - can you see it? Its name is \_\_pycache\_\_. Take a look inside. What do you see?

There is a file named (more or less) module.cpython-xy.pyc where x and y are digits derived from your version of Python (e.g., they will be 3 and 8 if you use Python 3.8).

The name of the file is the same as your module's name (module here). The part after the first dot says which Python implementation has created the file (CPython here) and its version number. The last part (pyc) comes from the words Python and compiled.

You can look inside the file - the content is completely unreadable to humans. It has to be like that, as the file is intended for Python's use only.

When Python imports a module for the first time, **it translates its contents into a somewhat compiled shape**.

The file doesn't contain machine code - it's internal Python **semi-compiled code**, ready to be executed by Python's interpreter. As such a file doesn't require lots of the checks needed for a pure source file, the execution starts faster, and runs faster, too. Thanks to that, every subsequent import will go quicker than interpreting the source text from scratch.

Python is able to check if the module's source file has been modified (in this case, the pyc file will be rebuilt) or not (when the pyc file may be run at once). As this process is fully automatic and transparent, you don't have to keep it in mind.

## 1.3.1.3 Modules and Packages: Your first module: step 3-6

Now we've put a little something into the module file:

print("I like to be a module.")

Can you notice any differences between a module and an ordinary script? There are none so far.

It's possible to run this file like any other script. Try it for yourself.

What happens? You should see the following line inside your console:

I like to be a module.

**Your first module: step 4**

Let's go back to the main.py file:

import module

Run it. What do you see? Hopefully, you see something like this:

I like to be a module.

What does it actually mean? When a module is imported, its content is **implicitly executed by Python**. It gives the module the chance to initialize some of its internal aspects (e.g., it may assign some variables with useful values).

Note: **the initialization takes place only once**, when the first import occurs, so the assignments done by the module aren't repeated unnecessarily.

Imagine the following context:

* there is a module named mod1;
* there is a module named mod2 which contains the import mod1 instruction;
* there is a main file containing the import mod1 and import mod2 instructions.

At first glance, you may think that mod1 will be imported twice - fortunately, **only the first import occurs**. Python remembers the imported modules and silently omits all subsequent imports.

**Your first module: step 5**

Python can do much more. It also creates a variable called \_\_name\_\_.

Moreover, each source file uses its own, separate version of the variable - it isn't shared between modules.

We'll show you how to use it. Modify the module a bit:

print("I like to be a module.")

print(\_\_name\_\_)

Now run the module.py file. You should see the following lines:

I like to be a module

\_\_main\_\_

Now run the main.py file. And? Do you see the same as us?

I like to be a module

module

We can say that:

* when you run a file directly, its \_\_name\_\_ variable is set to \_\_main\_\_;
* when a file is imported as a module, its \_\_name\_\_ variable is set to the file's name (excluding .py)

**Your first module: step 6**

This is how you can make use of the \_\_main\_\_ variable in order to detect the context in which your code has been activated:

if \_\_name\_\_ == "\_\_main\_\_":

print("I prefer to be a module.")

else:

print("I like to be a module.")

There's a cleverer way to utilize the variable, however. If you write a module filled with a number of complex functions, you can use it to place a series of tests to check if the functions work properly.

Each time you modify any of these functions, you can simply run the module to make sure that your amendments didn't spoil the code. These tests will be omitted when the code is imported as a module.

## 1.3.1.4 Modules and Packages: Your first module: step 7-10 12/28/21

**Your first module: step 7**

This module will contain two simple functions, and if you want to know how many times the functions have been invoked, you need a counter initialized to zero when the module is being imported. You can do it this way:

counter = 0

if \_\_name\_\_ == "\_\_main\_\_":

print("I prefer to be a module.")

else:

print("I like to be a module.")

**Your first module: step 8**

Introducing such a variable is absolutely correct, but may cause important side effects that you must be aware of. Take a look at the modified main.py file:

import module

print(module.counter)

As you can see, the main file tries to access the module's counter variable. Is this legal? Yes, it is. Is it usable? It may be very usable. Is it safe?

That depends - if you trust your module's users, there's no problem; however, you may not want the rest of the world to see your **personal/private variable**.

Unlike many other programming languages, Python has no means of allowing you to hide such variables from the eyes of the module's users. You can only inform your users that this is your variable, that they may read it, but that they should not modify it under any circumstances. This is done by preceding the variable's name with \_ (one underscore) or \_\_ (two underscores), but remember, it's only a convention. Your module's users may obey it or they may not.

Of course, we'll follow the convention. Now let's put two functions into the module - they'll evaluate the sum and product of the numbers collected in a list.In addition, let's add some ornaments there and remove any superfluous remnants.

**Your first module: step 9**

Okay. Let's write some brand new code in our module.py file. The updated module is ready here:

#!/usr/bin/env python3

""" module.py - an example of a Python module """

\_\_counter = 0

def suml(the\_list):

global \_\_counter

\_\_counter += 1

the\_sum = 0

for element in the\_list:

the\_sum += element

return the\_sum

def prodl(the\_list):

global \_\_counter

\_\_counter += 1

prod = 1

for element in the\_list:

prod \*= element

return prod

if \_\_name\_\_ == "\_\_main\_\_":

print("I prefer to be a module, but I can do some tests for you.")

my\_list = [i+1 for i in range(5)]

print(suml(my\_list) == 15)

print(prodl(my\_list) == 120)

A few elements need some explanation, we think:

* the line starting with #! has many names - it may be called shabang, shebang, hashbang, poundbang or even hashpling (don't ask us why). The name itself means nothing here - its role is more important. From Python's point of view, it's just a **comment** as it starts with #. For Unix and Unix-like OSs (including MacOS) such a **line instructs the OS how to execute the contents of the file** (in other words, what program needs to be launched to interpret the text). In some environments (especially those connected with web servers) the absence of that line will cause trouble;
* a string (maybe a multiline) placed before any module instructions (including imports) is called the **doc-string**, and should briefly explain the purpose and contents of the module;
* functions defined inside the module (suml() and prodl()) are available for import;
* we've used the \_\_name\_\_ variable to detect when the file is run stand-alone, and seized this opportunity to perform some simple tests.

**Your first module: step 10**

Now it's possible to use the updated module - this is one way:

from module import suml, prodl

zeroes = [0 for i in range(5)]

ones = [1 for i in range(5)]

print(suml(zeroes))

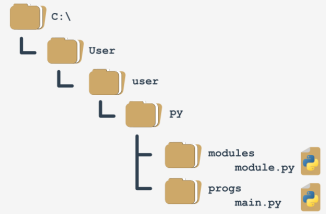
print(prodl(ones))

## 1.3.1.5 Modules and Packages: Your first module: step 11-12

**Your first module: step 11**

It's time to make our example more complicated - so far we've assumed that the main Python file is located in the same folder/directory as the module to be imported. Let's give up this assumption and conduct the following thought experiment:

* we are using Windows ® OS (this assumption is important, as the file name's shape depends on it)
* the main Python script lies in C:\Users\user\py\progs and is named main.py
* the module to import is located in C:\Users\user\py\modules

How to deal with it? To answer this question, we have to talk about **how Python searches for modules**. There's a special variable (actually a list) storing all locations (folders/directories) that are searched in order to find a module which has been requested by the import instruction.

Python browses these folders in the order in which they are listed in the list - if the module cannot be found in any of these directories, the import fails. Otherwise, the first folder containing a module with the desired name will be taken into consideration (if any of the remaining folders contains a module of that name, it will be ignored). The variable is named *path*, and it's accessible through the module named *sys*. This is how you can check its regular value:

import sys

for p in sys.path:

print(p)

We've launched the code inside the C:\User\user folder, and this is what we've got:

C:\Users\user

C:\Users\user\AppData\Local\Programs\Python\Python36-32\python36.zip

C:\Users\user\AppData\Local\Programs\Python\Python36-32\DLLs

C:\Users\user\AppData\Local\Programs\Python\Python36-32\lib

C:\Users\user\AppData\Local\Programs\Python\Python36-32

C:\Users\user\AppData\Local\Programs\Python\Python36-32\lib\site-packages

Note: the folder in which the execution starts is listed in **the first path's element**.

Note once again: there is a zip file listed as one of the path's elements - it's not an error. Python is able to treat zip files as ordinary folders - this can save lots of storage.

Can you figure out how we can solve our problem now? We can add a folder containing the module to the path variable (it's fully modifiable).

**Your first module: step 12**

One of several possible solutions looks like this:main.py

from sys import path

path.append('..\\modules')

import module

zeroes = [0 for i in range(5)]

ones = [1 for i in range(5)]

print(module.suml(zeroes))

print(module.prodl(ones))

Note:

* we've doubled the \ inside folder name - do you know why?

Check: Because a backslash is used to escape other characters - if you want to get just a backslash, you have to escape it.

* we've used the relative name of the folder - this will work if you start the main.py file directly from its home folder, and won't work if the current directory doesn't fit the relative path; you can always use an absolute path, like this:

path.append('C:\\Users\\user\\py\\modules')

* we've used the append() method - in effect, the new path will occupy the last element in the path list; if you don't like the idea, you can use insert() instead.

## 1.3.1.6 Modules and Packages: step 1-2

**Your first package: step 1**

Imagine that in the not-so-distant future you and your associates write a large number of Python functions. Your team decides to group the functions in separate modules, and this is the final result of the ordering:



alpha.py:

#! /usr/bin/env python3

""" module: alpha """

def funA():

return "Alpha"

if \_\_name\_\_ == "\_\_main\_\_":

print("I prefer to be a module.")

Note: we've presented the whole content for the alpha.py module only - assume that all the modules look similar (they contain one function named funX, where X is the first letter of the module's name).

**Your first package: step 2**

Suddenly, somebody notices that these modules form their own hierarchy, so putting them all in a flat structure won't be a good idea.

After some discussion, the team comes to the conclusion that the modules have to be grouped. All participants agree that the following tree structure perfectly reflects the mutual relationships between the modules:



Let's review this from the bottom up:

* the ugly group contains two modules: psi and omega;
* the best group contains two modules: sigma and tau;
* the good group contains two modules (alpha and beta) and one subgroup (best)
* the extra group contains two subgroups (good and bad) and one module (iota)

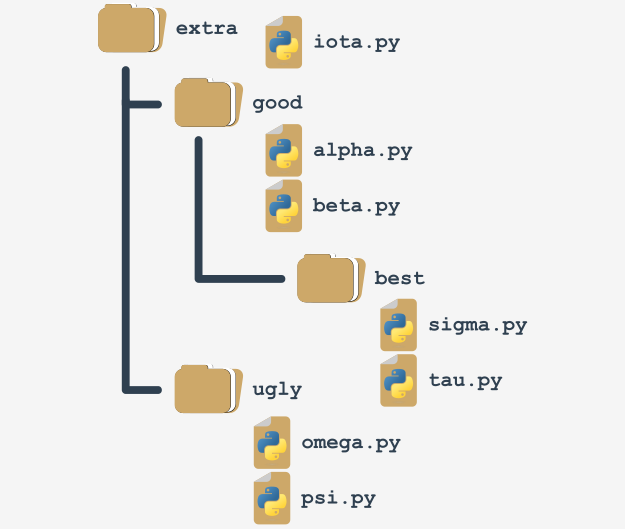
Does it look bad? Not at all - analyze the structure carefully. It resembles something, doesn't it? It looks like a **directory structure**.

Let's build a tree reflecting projected dependencies between the modules.

## 1.3.1.7 Modules and Packages: step 3-4

**Your first package: step 3**

This is how the tree currently looks:

Such a structure is almost a package (in the Python sense). It lacks the fine detail to be both functional and operative. We'll complete it in a moment.

If you assume that extra is the name of a newly created package (think of it as the package's root), it will impose a naming rule which allows you to clearly name every entity from the tree. For example:

* the location of a function named funT() from the tau package may be described as: extra.good.best.tau.funT()
* a function marked as: extra.ugly.psi.funP()

comes from the psi module being stored in the ugly subpackage of the extra package.

**Your first package: step 4**

There are two questions to answer:

* **how** do you transform such a tree (actually, a subtree) into a real Python **package** (in other words, how do you convince Python that such a tree is not just a bunch of junk files, but a set of modules)?
* **where** do you put the subtree to make it accessible to Python?

The first question has a surprising answer: **packages, like modules, may require initialization**.

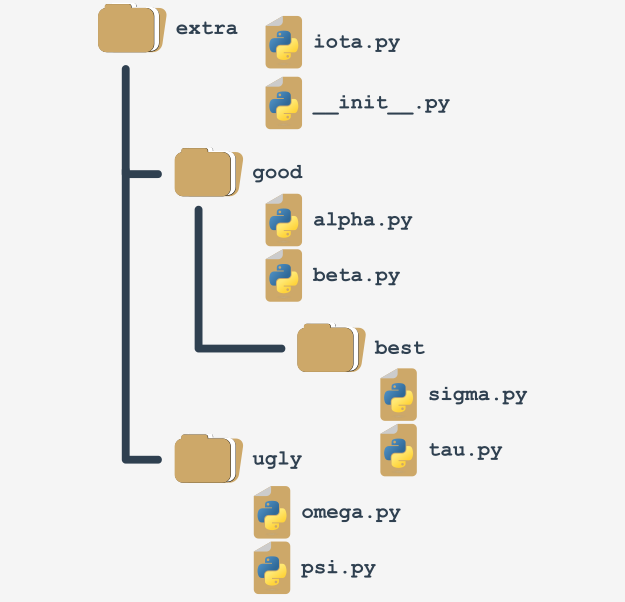
The initialization of a module is done by an unbound code (not a part of any function) located inside the module's file. As a package is not a file, this technique is useless for initializing packages. You need to use a different trick instead - Python expects that there is a file with a very unique name inside the package's folder: \_\_init\_\_.py.

The content of the file is executed when any of the package's modules is **imported**. If you don't want any special initializations, you can leave the file empty, but you mustn't omit it.

## 1.3.1.8 Modules and Packages: step 5-6

**Your first package: step 5**

Remember: **the presence of the \_\_init.py\_\_ file finally makes up the package.**

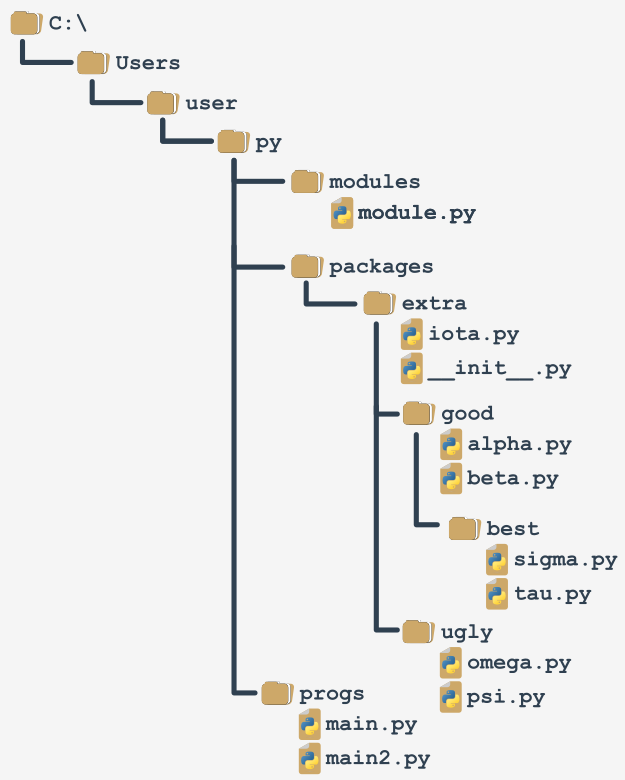
Note: it's not only the root folder that can contain \_\_init.py\_\_ file - you can put it inside any of its subfolders (subpackages) too. It may be useful if some of the subpackages require individual treatment and special kinds of initialization.

Now it's time to answer the second question - the answer is simple: **anywhere**. You only have to ensure that Python is aware of the package's location.

You already know how to do that. You're ready to make use of your first package.

**Your first package: step 6**

Let's assume that the working environment looks as follows:

We've prepared a zip file containing all the files from the packages branch. You can download it and use it for your own experiments, but remember to unpack it in the folder presented in the scheme, otherwise, it won't be accessible to the code from the main file.

DOWNLOAD Modules and Packages ZIP file

You'll be continuing your experiments using the main2.py file.

## 1.3.1.9 Modules and Packages: step 7

**Your first package: step 7**

We will access the funI() function from the iota module from the top of the extra package. It forces us to use qualified package names (associate this with naming folders and subfolders - the conventions are very similar). This is how to do it:

# main2.py

from sys import path

path.append('..\\packages')

import extra.iota

print(extra.iota.funI())

Note:

* we've modified the path variable to make it accessible to Python;
* the import doesn't point directly to the module, but specifies the fully qualified path from the top of the package;

replacing import extra.iota with import iota will cause an error. The following variant is valid too:

from sys import path

path.append('..\\packages')

from extra.iota import funI

print(funI())

Note the qualified name of the iota module.

## 1.3.1.10 Modules and Packages: step 8-9

**Your first package: step 8**

Now let's reach all the way to the bottom of the tree - this is how to get access to the *sigma* and *tau* modules:

from sys import path

path.append('..\\packages')

import extra.good.best.sigma

from extra.good.best.tau import funT

print(extra.good.best.sigma.funS())

print(funT())

You can make your life easier by using aliasing:

from sys import path

path.append('..\\packages')

import extra.good.best.sigma as sig

import extra.good.alpha as alp

print(sig.funS())

print(alp.funA())

**Your first package: step 9**

Let's assume that we've zipped the whole subdirectory, starting from the extra folder (including it), and let's get a file named extrapack.zip. Next, we put the file inside the packages folder. Now we are able to use the zip file in a role of packages:

from sys import path

path.append('..\\packages\\extrapack.zip')

import extra.good.best.sigma as sig

import extra.good.alpha as alp

from extra.iota import funI

from extra.good.beta import funB

print(sig.funS())

print(alp.funA())

print(funI())

print(funB())

If you want to conduct your own experiments with the package we've created, you can download it below. We encourage you to do so.

DOWNLOAD Extrapack ZIP file

Now you can create modules and combine them into packages. It's time to start a completely different discussion - about errors, failures and crashes.

## 1.3.1.11 SECTION SUMMARY

1. While a **module** is designed to couple together some related entities (functions, variables, constants, etc.), a **package** is a container which enables the coupling of several related modules under one common name. Such a container can be distributed as-is (as a batch of files deployed in a directory sub-tree) or be packed into a zip file.

2. During the very first import of the actual module, Python translates its source code into the **semi-compiled** format stored inside the **pyc** files, and deploys these files into the \_\_pycache\_\_ directory located in the module's home directory.

3. If you want to instruct your module's user that a particular entity should be treated as **private** (i.e. not to be explicitly used outside the module) you can mark its name with either the \_ or \_\_ prefix. Don't forget that this is only a recommendation, not an order.

4. The names shabang, shebang, hasbang, poundbang, and hashpling describe the digraph written as #!, used to instruct Unix-like OSs how the Python source file should be launched. This convention has no effect under MS Windows.

5. If you want convince Python that it should take into account a non-standard package's directory, its name needs to be inserted/appended into/to the import directory list stored in the path variable contained in the sys module.

1. A Python file named \_\_init\_\_.py is implicitly run when a package containing it is subject to import, and is used to initialize a package and/or its sub-packages (if any). The file may be empty, but must not be absent.

**Exercise 1**

You want to prevent your module's user from running your code as an ordinary script. How will you achieve such an effect? Check

import sys

if \_\_name\_\_ == "\_\_main\_\_":

print "Don't do that!"

sys.exit()

**Exercise 2**

Some additional and necessary packages are stored inside the D:\Python\Project\Modules directory. Write a code ensuring that the directory is traversed by Python in order to find all requested modules. Check

import sys

# note the double backslashes!

sys.path.append("D:\\Python\\Project\\Modules")

**Exercise 3**

The directory mentioned in the previous exercise contains a sub-tree of this structure:

abc

|\_\_ def

|\_\_ mymodule.py

Assuming that D:\Python\Project\Modules has been successfully appended to the sys.path list, write an import directive letting you use all the mymodule entities.

Check: import abc.def.mymodule

# 1.4. PYTHON PACKAGE INSTALLER (PIP)

## **1.4.1.1 Python Package Installer (PIP): packaging ecosystem**

**Python packaging ecosystem and how to use it**

Python is a very powerful instrument – we hope you've experienced this yourself already. Many people from around the world feel this way, and they use Python on a regular basis to develop what they can do in many completely different fields of activity. This means that Python has become an interdisciplinary tool employed in countless applications. We can’t go through all the spheres in which Python brilliantly shows off its abilities, so let us just tell you about the most impressive ones.

First of all, Python has turned into a leader of research on artificial intelligence. Data mining, one of the most promising modern scientific disciplines, utilizes Python as well. Mathematicians, psychologists, geneticists, meteorologists, linguists – all these people already use Python, or if they don’t already, we’re sure that they will very soon. There is no escaping this trend.

Of course, it doesn’t make any sense to get all Python users to write their code from scratch, keeping them perfectly isolated from the outside world and from other programmers' achievements. This would be both unnatural and counterproductive.

The most preferable and efficient thing is to enable all Python community members to freely exchange their codes and experiences. In this model, nobody is forced to start work from scratch, as there’s a high probability that someone else has been working on the same (or a very similar) problem.

As you know, Python was created as open-source software, and this also works as an invitation for all coders to maintain the whole Python ecosystem as an open, friendly, and free environment. To make the model work and evolve, some additional tools should be provided, tools that help the creators to publish, maintain, and take care of their code.

These same tools should help users to make use of the code, both the already existing code, and also the new code appearing every day. Thanks to that, writing new code for new challenges is not like building a new house, starting at the foundations.

Moreover, the programmer is free to modify someone else's code in order to adapt it to their own needs, and in effect build a completely new product that can be used by another developer. The process seems to have no end. Fortunately.

To make this world go round, two basic entities have to be established and kept in motion: a **centralized repository** of all available software packages; and a tool allowing users to **access the repository**. Both these entities already exist and can be used at any time.

## **1.4.1.2 Python Package Installer (PIP): packaging ecosystem**

**Python packaging ecosystem and how to use it: continued**

The repository (or repo for short) we mentioned before is named PyPI (it's short for Python Package Index) and it's maintained by a workgroup named the Packaging Working Group, a part of the Python Software Foundation, whose main task is to support Python developers in efficient code dissemination.

You can find their website here: <https://wiki.python.org/psf/PackagingWG.>

The PyPI website (administered by PWG) is located at the address: <https://pypi.org/.>

When we popped in there for a while at the beginning of June 2020, we found that PyPI was home to 237,515 projects, consisting of 2,877,545 files managed by 427,487 users.

These three numbers alone clearly show the potency of the Python community and the importance of developer cooperation.

We must point out that PyPI is not the only existing Python repository. On the contrary, there are lots of them, created for projects and led by many larger and smaller Python communities. It's likely that someday you and your colleagues may want to create your own repos.

Anyway, PyPI is the most important Python repo in the world. If we modify the classic saying a little, we can state that “all Python roads lead to PyPl”, and that’s no exaggeration at all.

## **1.4.1.3 Python Package Installer (PIP): The PyPI repo**

**The PyPI repo: the Cheese Shop**

The PyPI repo is sometimes referred to as the Cheese Shop. Really.

Does that sound a little strange to you? Don't worry, it’s all perfectly innocent.

We refer to the repo as a shop, because you go there for the same reasons you go to other shops: to fulfill your needs. If you want some cheese, you go to the cheese shop. If you want a piece of software, you go to the software shop. Fortunately, the analogy ends here. you don't need money to take some software out of the repo shop.

PyPI is completely free, and you can just pick a code and use it – you’ll encounter neither cashier nor security guard. Of course, it doesn't absolve you from being polite and honest. You have to obey all the licensing terms, so don't forget to read them.

“OK”, you say, “the shop is clear now, but what does cheese have to do with Python?”

The Cheese Shop is one of the most famous Monty Python sketches. It depicts the surrealist adventure of an Englishman trying to buy some cheese. Unfortunately, the shop he visits (immodestly named Ye National Cheese Emporium) has no cheese in stock at all.

Of course, it's meant to be ironic. As you already know, PyPI has lots of software in stock and it's available 24/7. It's fully entitled to identify itself as Ye International Python Software Emporium.

## **1.4.1.4 Python Package Installer (PIP):** The PyPI repo

**The PyPI repo: the Cheese Shop (continued)**

PyPI is a very specific shop, not just because it offers all its products for free. It also requires a special tool to make use of it. Fortunately, this tool is also free, so if you want to make your own digital cheeseburger by using the goods offered by the PyPI Shop, you’ll need a free tool named pip.

No, you haven't misheard. Just pip. It's another acronym, of course, but its nature is more complex than the previously mentioned PyPI, as it's an example of a recursive acronym, which means that the acronym refers to itself, which means that explaining it is an infinite process.

Why? Because pip means “pip installs packages”, and the pip inside “pip installs packages” means “pip installs packages” and ... Let’s stop there. Thank you for your cooperation. By the way, there are a few other very famous recursive acronyms. One of them is Linux, which can be interpreted as “Linux is Not Unix”.

## **1.4.1.5 Python Package Installer (PIP): How to install pip**

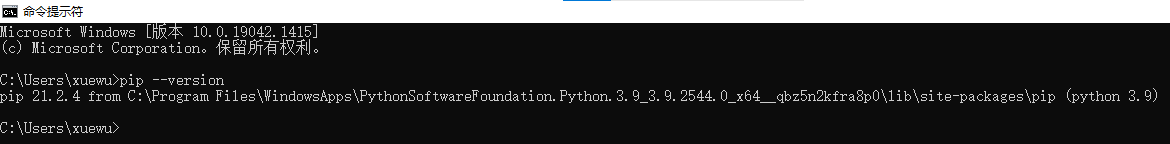
**How to install pip**

The question that should be put now is: how to get a proper cheese knife? In other words, how to ensure that pip is installed and ready to work? The most precise answer is “it depends”. Really. Some Python installations come with pip, some don't. What’s more, it doesn’t only depend on the OS you use, although this is a very important factor. Let's start with MS Windows.

## **1.4.1.6 pip installation on MS Windows**

**pip on MS Windows**

The MS Windows Python installer already contains pip, and so no other steps need to be taken in order to install it. Unfortunately, if the PATH variable is misconfigured, pip may unavailable. To verify that we haven’t misled you, try to do this:

* open the Windows console (CMD or PowerShell, whatever you prefer)
* execute the following command: pip --version
* in the most optimistic scenario (and we really want that to happen) you’ll see something like this:
* 
* the absence of this message may mean that the PATH variable either incorrectly points to the location of the Python binaries, or doesn't point to it at all; for example, our PATH variable contains the following substring:

C:\Program Files\Python3\Scripts\;C:\Program Files\Python3\;

* the easiest way to reconfigure the PATH variable is to **reinstall Python**, instructing the installer to set it for you.

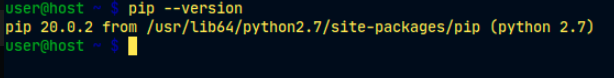
## **1.4.1.7 pip installation on Linux**

**pip on Linux**

Different Linux distributions may behave differently when it comes to using pip. Some of them (like Gentoo), which are closely bound to Python and which use it internally, may have pip preinstalled and are instantly ready to work.

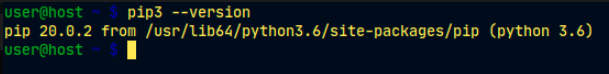
Don't forget that some Linuces may utilize more than one Python version concurrently, e.g., one Python 2 and one Python 3 coexisting side by side. Such systems may launch Python 2 as the default version, and it may be necessary to explicitly specify the program name as python3. In this case there may be two different pips identified as pip (or pip2) and pip3. Check it carefully.

Open the terminal window and issue the following command: pip --version



An answer similar to the one shown in the previous picture determines that you've launched pip from Python 2, so the next try should look as follows:

pip3 --version



As you can see, we’re now sure that we’re using the appropriate version of pip.

## **1.4.1.8 pip installation on Linux**

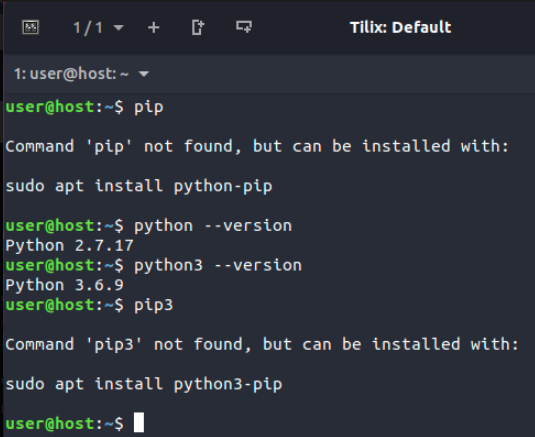
**pip on Linux: continued**

Unfortunately, some Linux distributions don't have pip preinstalled, even if Python itself is installed by default (some versions of Ubuntu may behave this way). In this case, you have two possibilities:

* install pip as a system package using a dedicated package manager (e.g., apt in Debian-like systems)
* install pip using internal Python mechanisms.

The former is definitely better. Although there are some smart scripts that are able to download and install pip by ignoring the OS, we discourage you from using them. This method can get you into trouble.

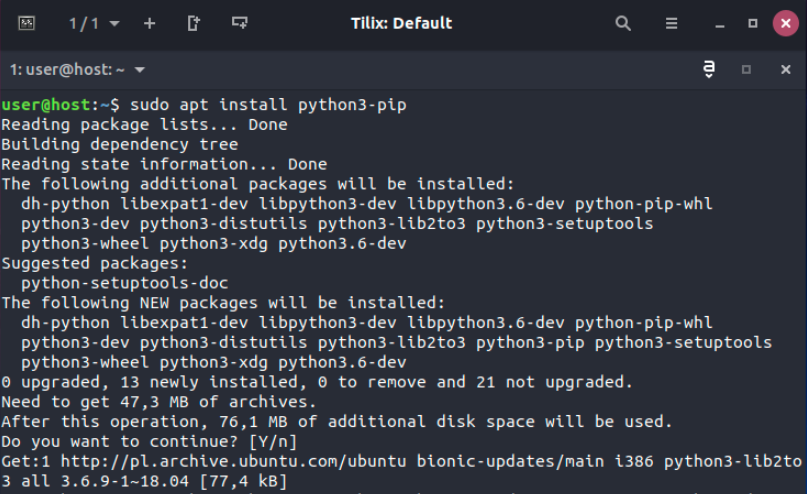
Look – we've tried to launch pip3 and we’ve failed. Our OS (we used Ubuntu Budgie this time) suggested using apt in order to install the package named python3-pip:



That's good advice, and we're going to follow it, but it has to be stated that we’ll need administrative rights to do it. Don't forget that different Linuces may use different package managers (e.g., it could be pacman if you use Arch Linux, or yum used by distributions derived from Red Hat).

Anyway, all these methods should get pip (or pip3) installed and working.

Look what happened when we followed the OS suggestion:

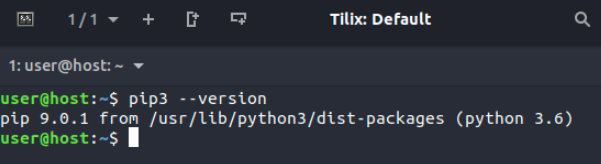


As you can see, the OS decided to install not only pip itself, but also a couple of additional components needed by pip. This is normal – don't be alarmed.

## **1.4.1.9 pip installation on Linux**

**pip on Linux: continued**

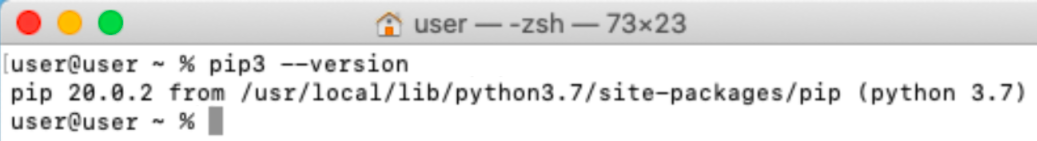
When apt finishes its job, we are finally able to utilize pip3:



If you're a Mac user and you've installed Python 3 using the brew installer, pip is already present in your system and ready to work. Check it by issuing the previously mentioned command:

pip3 --version

and wait for the response. This is what we saw:



## **1.4.1.10 Dependencies**

**Dependencies**

Now that we’re sure that pip is ready at our command, we’re going to limit our focus to MS Windows only, as its behavior is (should be) the same in all OSes, but before we start, we need to explain an important issue and tell you about **dependencies**.

Imagine that you've created a brilliant Python application named redsuspenders, able to predict stock exchange rates with 99% accuracy (by the way, if you actually do that, please contact us immediately).

Of course, you've used some existing code to achieve this goal – e.g., your app imports a package named nyse containing some crucial functions and classes. Moreover, the nyse package imports another package named wallstreet, while the wallstreet package imports other two essential packages named *bull* and *bear*.

As you’ve probably already guessed, the connections between these packages are crucial, and if somebody decides to use your code (but remember, we've already called dibs on it) they will also have to ensure that all required packages are in place.

To make a long story short, we can say that **dependency is a phenomenon that appears every time you're going to use a piece of software that relies on other software.** Note that dependency may include (and generally does include) more than one level of software development.

Does this mean that a potential nyse package user is obliged to trace all dependencies and manually install all the needed packages? That would be horrible, wouldn't it?

Yes, it's definitely horrible, so you shouldn't be surprised that the process of arduously fulfilling all the subsequent requirements has its own name, and it's called dependency hell.

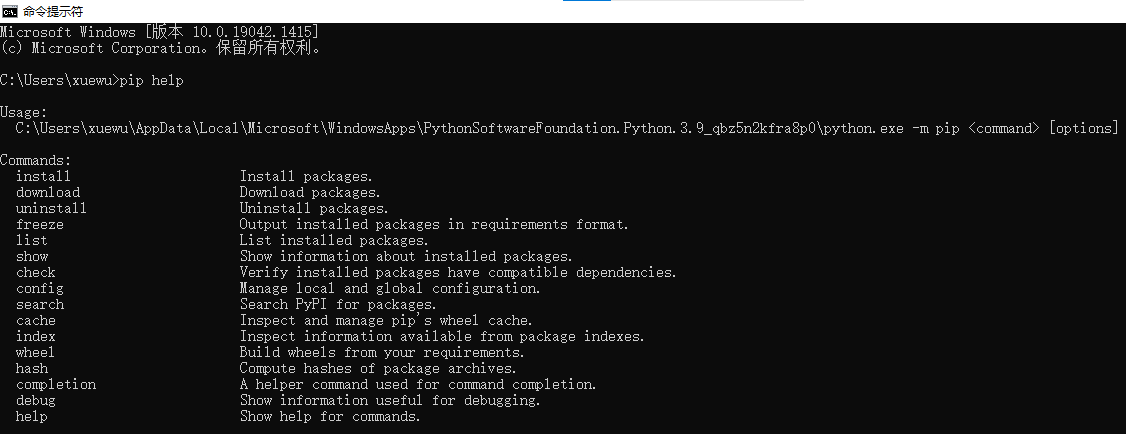
How do we deal with that? Is every user doomed to visit hell in order to run the code for the first time?

Fortunately not - pip can do all of this for you. Really. It can discover, identify, and resolve all dependencies. Moreover, it can do it in the cleverest way, avoiding any unnecessary downloads and reinstalls.

## **1.4.1.11 How to use pip: How to use pip**

Now we’re ready to ask pip what it can do for us. Let's do it – issue the following command: pip help

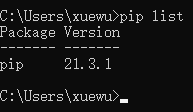
and wait for pip's response. This is what it looks like:



Don't forget that you may be obliged to replace pip with pip3 if your environment requires this.The list produced by pip summarizes all the available operations, and the last of them is help, which we've just used already.If you want to know more about any of the listed operations, you can use the following form of pip invocation: pip help operation. For example, the line: *pip help install* will show you detailed information about using and parameterizing the install command.

If you want to know what Python packages have been installed so far, you can use the list operation–just like this: pip list

The output you’ll see is rather unpredictable. Don't be surprised if your screen ends up being filled with completely different content. Ours look as follows:

As you can see, there are two columns in the list, one showing the name of the installed package, and the other showing the version of the package. We can’t predict the state of your Python installation.

The only thing we know for sure is that your list contains the two lines we see on our list: pip and setuptools. This happens because the OS is convinced that a user wanting pip will very likely need the setuptools soon. It’s not wrong.

## **1.4.1.12 How to use pip: How to use pip: continued**

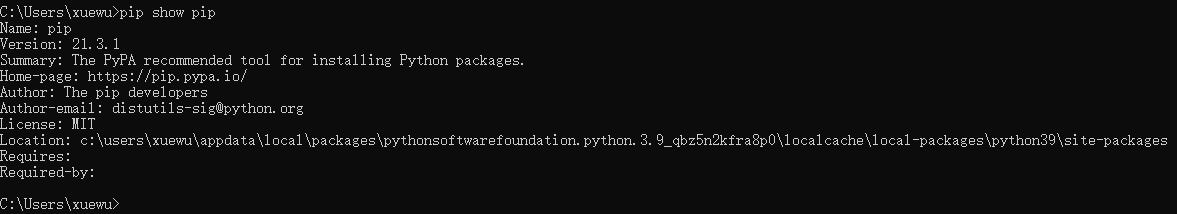
The pip list isn't very informative, and it may happen that it won't satisfy your curiosity. Fortunately, there’s a command that can tell you more about any of the installed packages (note the word installed). The syntax of the command looks like:

pip show package\_name

We’re going to use it in a slightly deceptive way – we want to convince pip to confess something about itself. This is how we do it:

pip show pip

It looks a bit odd, doesn't it? Despite this, it works fine, and pip's self-presentation looks consistent and current:



You may ask where this data comes from? Is pip really so perceptive? Not at all–the information appearing on the screen is taken from inside the package being shown. In other words, the package's creator is obliged to equip it with all the needed data (or to express it more precisely–metadata).

Look at the two lines at the bottom of the output. They show:

* which packages are needed to successfully utilize the package (*Requires*:)
* which packages need the package to be successfully utilized (*Required*-*by*:)

As you can see, both properties are empty. Feel free to try to use the show command in relation to any other installed package.

The power of pip comes from the fact that it’s actually a gateway to the Python software universe. Thanks to that, you can browse and install any of the hundreds of ready-to-use packages gathered in the PyPI repositories. Don't forget that pip is not able to store all PyPI content locally (it’s unnecessary and it would be uneconomical).

In effect, pip uses the Internet to query PyPI and to download the required data. This means that you have to have a network connection working whenever you’re going to ask pip for anything that may involve direct interactions with the PyPI infrastructure.

One of these cases occurs when you want to search through PyPI in order to find a desired package. This kind of search is initiated by the following command:

pip search anystring

The anystring provided by you will be searched in:

* the names of all the packages;
* the summary strings of all the packages.

Be aware of the fact that some searches may generate a real avalanche of data, so try to be as specific as possible. For example, an innocent-looking query like this one:

pip search pip

produces more than 100 lines of results (try it yourself – don't take our word for it). By the way – the search is case insensitive.

If you’re not a fan of console reading, you can use the alternative way of browsing PyPI content offered by a search engine, available at <https://pypi.org/search.>

## **1.4.1.13 How to use pip: How to use pip: continued**

Assuming that your search is successful (or you’re determined to install a specific package of an already known name) you can use pip to install the package onto your computer. Two possible scenarios may be put into action now:

* you want to install a new package for you only – it won't be available for any other user (account) existing on your computer; this procedure is the only one available if you can’t elevate your permissions and act as a system administrator;
* you’ve decided to install a new package system-wide – you have administrative rights and you're not afraid to use them.

To distinguish between these two actions, pip uses a dedicated option named --user (note the double dash). The presence of this option instructs pip to act locally on behalf of your (non-administrative) user.

If you don’t add this, pip assumes that you’re as a system administrator and it’ll do nothing to correct you if you’re not.

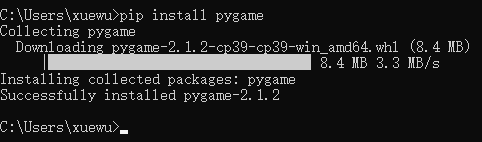
In our case, we’re going to install a package named pygame – it's an extended and complex library allowing programmers to develop computer games using Python.

The project has been in development since the year 2000, so it's a mature and reliable piece of code. If you want to know more about the project and about the community which leads it, visit https://www.pygame.org.

If you’re a system administrator, you can install pygame using the following command: pip install pygame

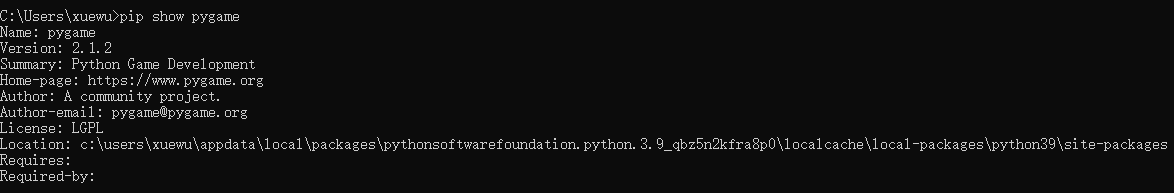
If you're not an admin, or you don't want to fatten up your OS by installing pygame system-wide, you can install it for you only: pip install --user pygame

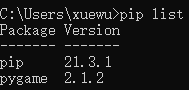
It's up to you which of the above procedures you want to take place.



Pip has a habit of displaying fancy textual animation indicating the installation progress, so watch the screen carefully – don't miss the show! If the process is successful, you’ll see something like this:

We encourage you to use: *pip show pygame* and *pip list* to get more information about what actually happened.





## **1.4.1.14 How to use pip: How to use pip: a simple test program**

Now that pygame is finally accessible, we can try to use it in a very simple test program. Let’s comment on it briefly.

line 1: import pygame and let it serve us;

line 3: the program will run as long as the run variable is True;

lines 4 and 5: determine the window's size;

line 6: initialize the pygame environment;

line 7: prepare the application window and set its size;

line 8: make an object representing the default font of size 48 points;

line 9: make an object representing a given text – the text will be anti-aliased (True) and white (255,255,255)

line 10: insert the text into the (currently invisible) screen buffer;

line 11: flip the screen buffers to make the text visible;

line 12: the pygame main loop starts here;

line 13: get a list of all pending pygame events;

lines 14 through 16: check whether the user has closed the window or clicked somewhere inside it or pressed any key;

line 15: if yes, stop executing the code.

import pygame

run = True

width = 400

height = 100

pygame.init()

screen = pygame.display.set\_mode((width, height))

font = pygame.font.SysFont(None, 48)

text = font.render("Welcome to pygame", True, (255, 255, 255))

screen.blit(text, ((width - text.get\_width()) // 2, (height - text.get\_height()) // 2))

pygame.display.flip()

while run:

for event in pygame.event.get():

if event.type == pygame.QUIT\

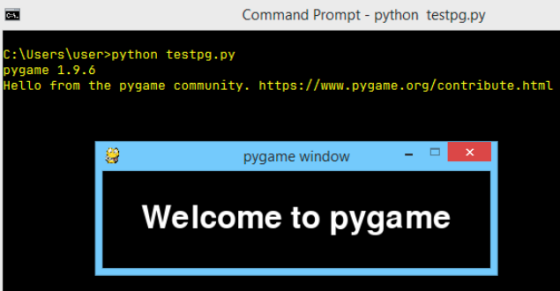
or event.type == pygame.MOUSEBUTTONUP\

or event.type == pygame.KEYUP:

run = False

## **1.4.1.15 How to use pip: How to use pip: continued**

This is what we expect from our impressive code:



The pip install has two important additional abilities:

* it is able to **update** a locally installed package – e.g., if you want to make sure that you’re using the latest version of a particular package, you can run the following command:

pip install -U package\_name

where -U means update. Note: this form of the command makes use of the --user option for the same purpose as presented previously;

* it is able to **install a user-selected version** of a package (pip installs the **newest** available version by default); to achieve this goal you should use this syntax:

pip install package\_name==package\_version

(note the double equals sign) e.g., pip install pygame==1.9.2

## **1.4.1.16 How to use pip: How to use pip: continued**

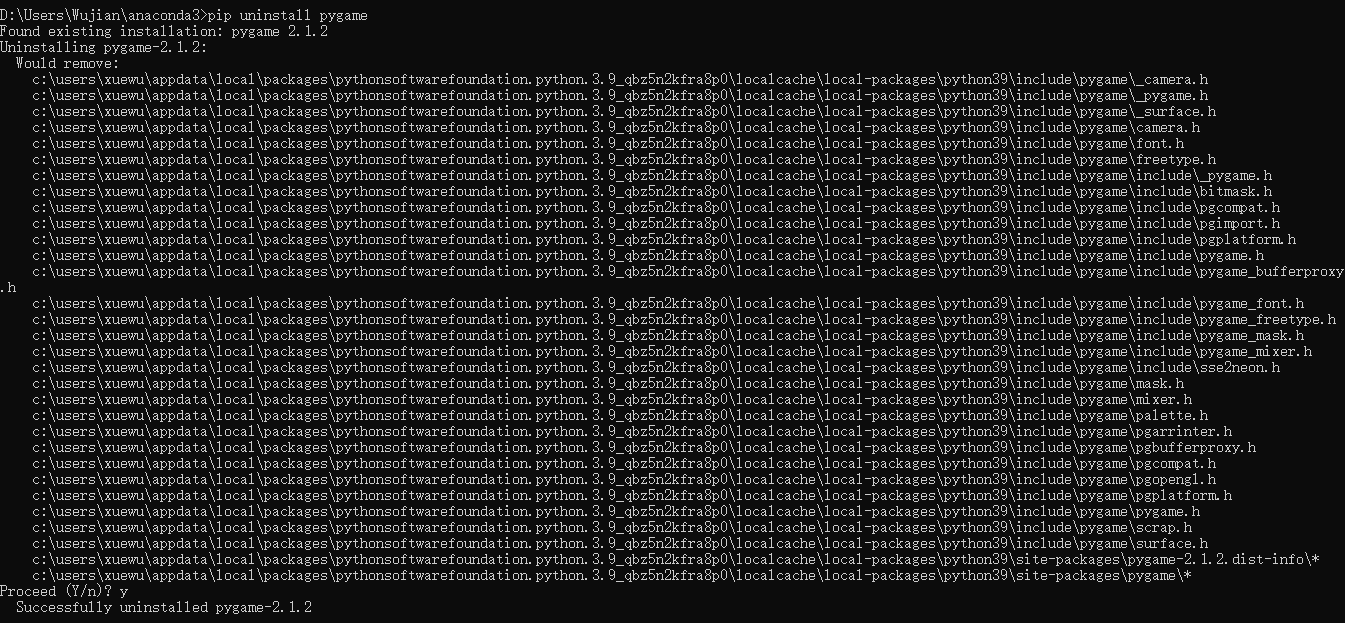
If any of the currently installed packages are **no longer needed** and you want to get rid of them, pip will be useful, too. Its *uninstall* command will execute all the needed steps. The required syntax is clear and simple:

*pip uninstall package\_name*

so if you don't want pygame anymore you can execute the following command:

*pip uninstall pygame*

Pip will want to know if you’re sure about the choice you're making – be prepared to give the right answer. The process looks like this:



## **1.4.1.17 How to use pip: Use pip!**

Pip's capabilities don't end here, but the command set we've presented to you is enough to start successfully managing packages that aren't a part of the regular Python installation.

We hope we’ve encouraged you to carry out your own experiments with pip and the Python package universe. PyPI invites you to dive into its extensive resources.

Some say that one of the most important programming virtues is **laziness**. Don't get us wrong – we don't want you to spend all day napping on the couch and dreaming of Python code.

A lazy programmer is a programmer who looks for existing solutions and analyzes the available code before they start to develop their own software from scratch. This is why PyPI and pip exist – use them!

## **1.4.1.18 SECTION SUMMARY: Key takeaways**

1. A **repository** (or **repo** for short) designed to collect and share free Python code exists and works under the name **Python Package Index** (PyPI) although it's also likely that you come across a very niche name The Cheese Shop. The Shop's website is available at <https://pypi.org/.>

2. To make use of The Cheese Shop the specialized tool has been created and its name is **pip** (pip installs packages while pip stands for... ok, don't mind). As pip may not be deployed as a part of standard Python installation, it is possible that you will need to install it manually. Pip is a console tool.

3. To check pip's version one the following commands should be issued:

pip --version or pip3 --version

Check yourself which of these works for you in your OS' environment.

4. List of main pip activities looks as follows:

* *pip help operation* - shows brief pip's description;
* *pip list* - shows list of currently installed packages;
* *pip show package\_name* - shows package\_name info including package's dependencies;
* *pip search anystring* - searches through PyPI directories in order to find packages which name contains anystring;
* *pip install name* - installs name system-wide (expect problems when you don't have administrative rights);
* *pip install --user name* - install name for you only; no other your platform's user will be able to use it;
* *pip install -U name* - updates previously installed package;
* *pip uninstall name* - uninstalls previously installed package;

**Exercise 1:** Where does the name "The Cheese Shop" come from?

Check: It's a reference to an old Monty Python's sketch of the same name.

**Exercise 2**: Why should I ensure which one of pip and pip3 works for me?

Check: When Python 2 and Python 3 coexist in your OS, it's likely that pip identifies the instance of pip working with Python 2 packages only.

**Exercise 3:** How can I determine if my pip works with either Python 2 or Python 3?

Check: pip --version will tell you that.

**Exercise 4:** Unfortunately, I don't have administrative right. What should I do to install a package system-wide?

Check: You have to ask your sysadmin - don't try to hack your OS!

## **1.4.1.19 Module Completion**

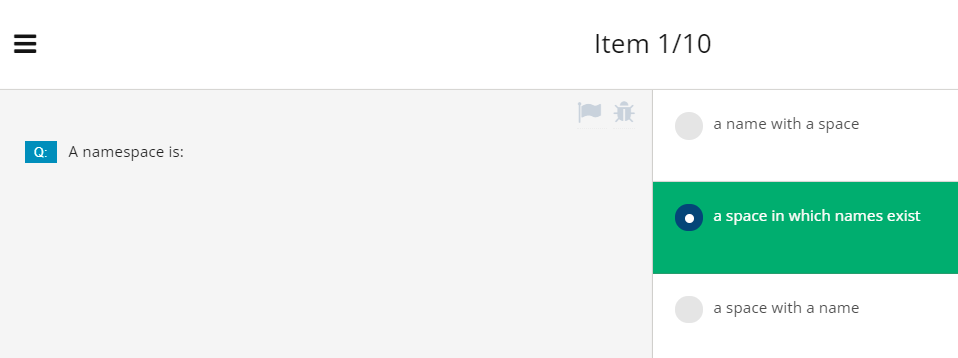
**Congratulations! You have completed PE2: Module 1.**

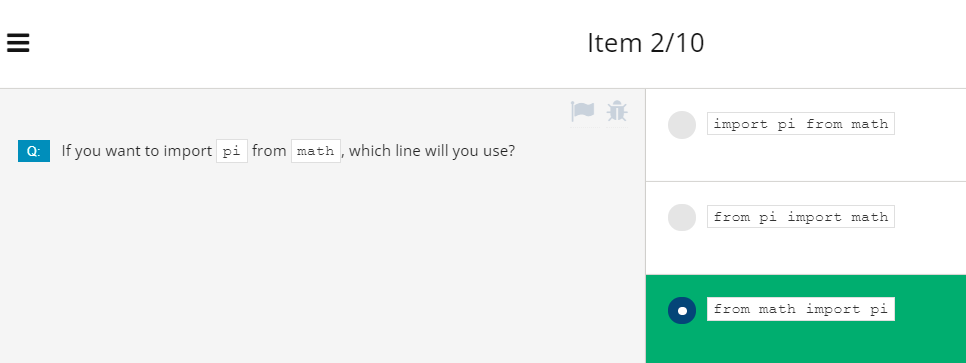
Well done! You've reached the end of Module 1 and completed a major milestone in your Python programming education. Here's a short summary of the objectives you've covered and got familiar with in Module 1:

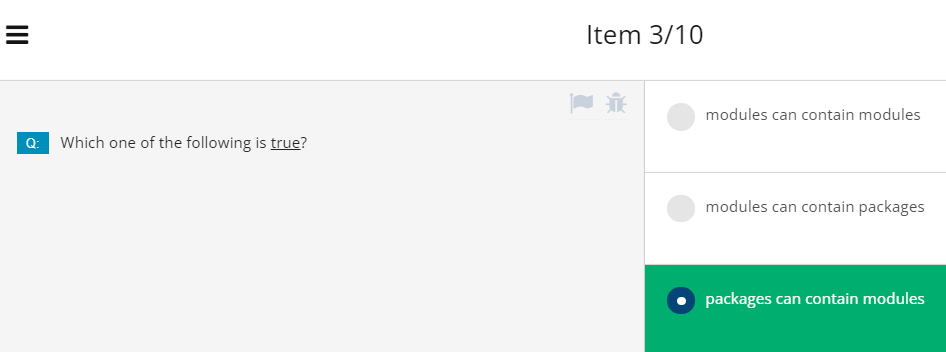
* working with Python modules; importing, creating, and using modules;
* using selected Python STL modules (math, random, and platform)
* constructing and using packages in Python;
* PIP (Python Package Installer.)

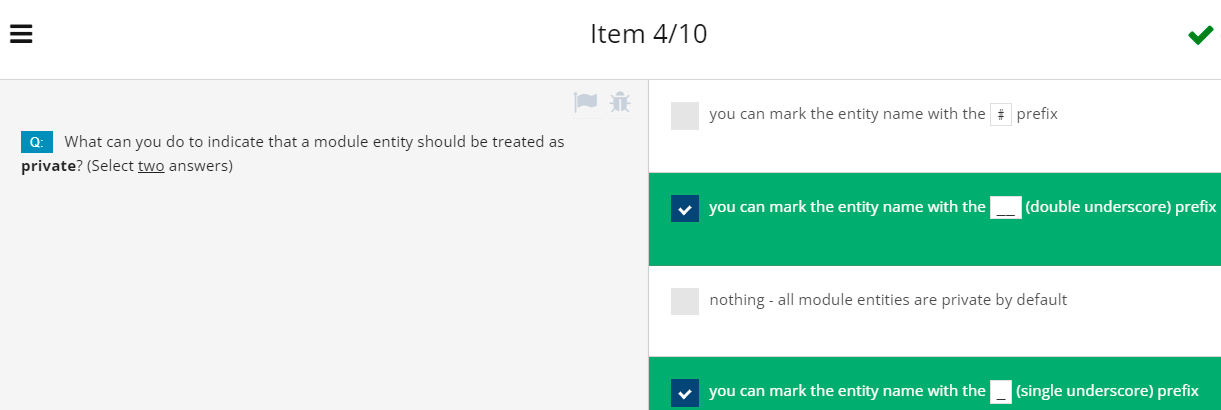
You are now ready to take the module quiz and attempt the final challenge: Module 1 Test, which will help you gauge what you've learned so far.

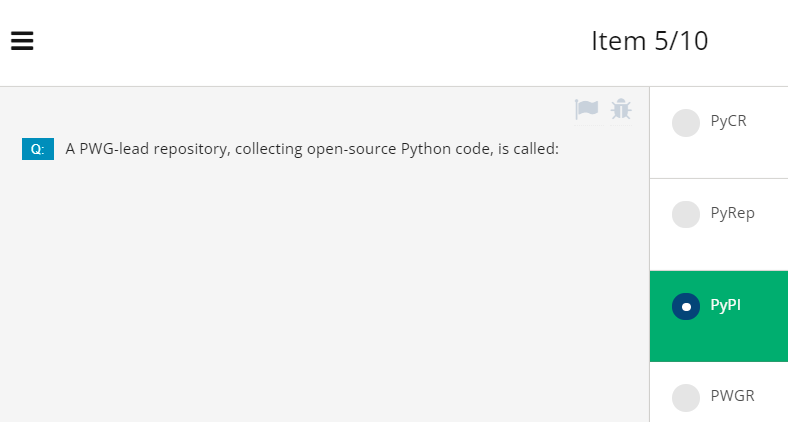
# 1.5. **PE2 -- Module 1 Quiz**

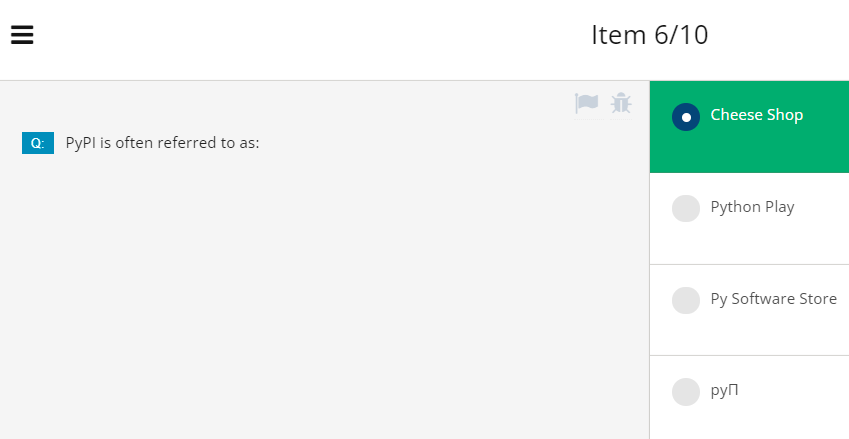


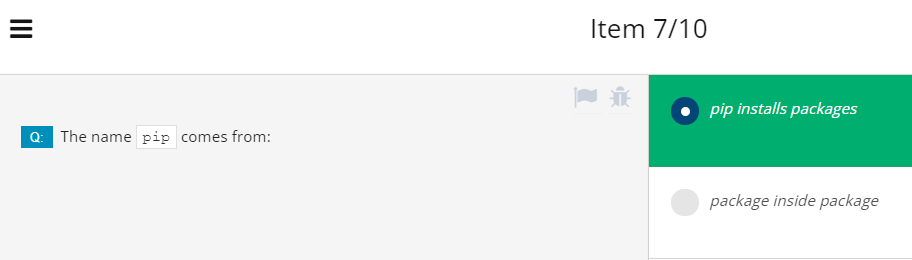


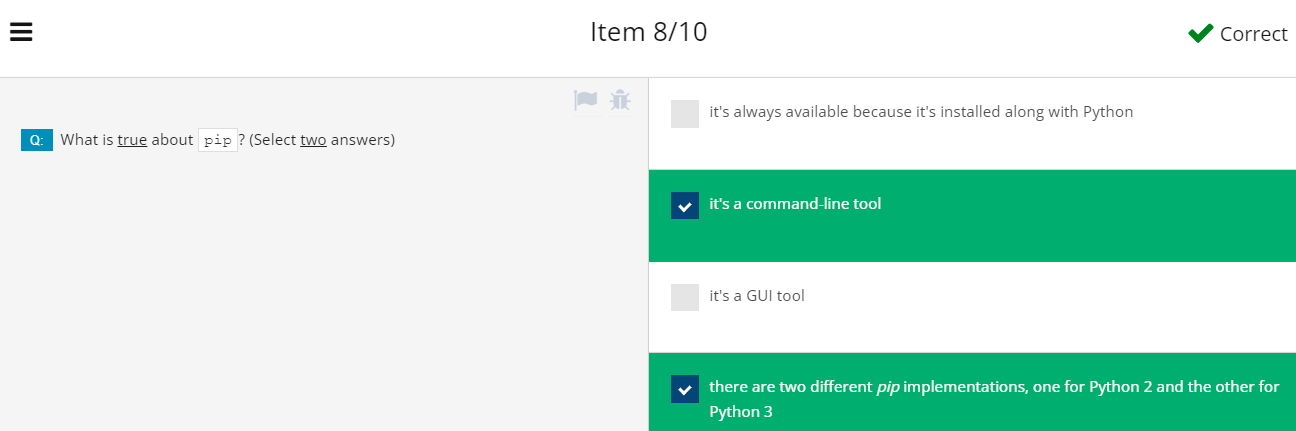


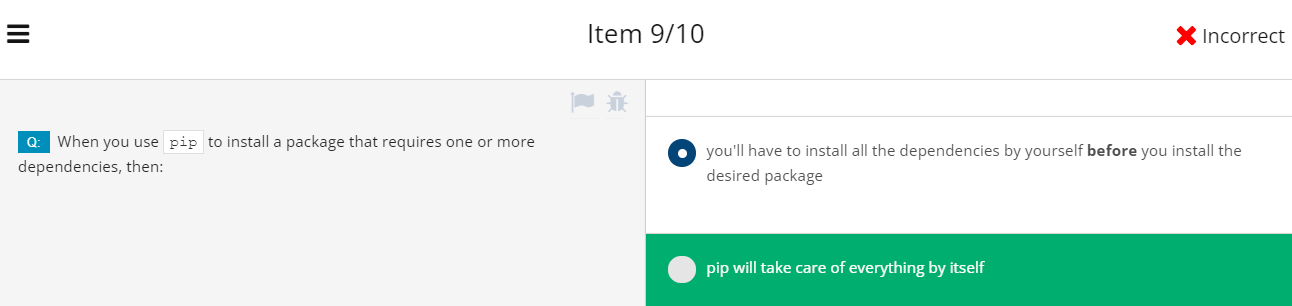


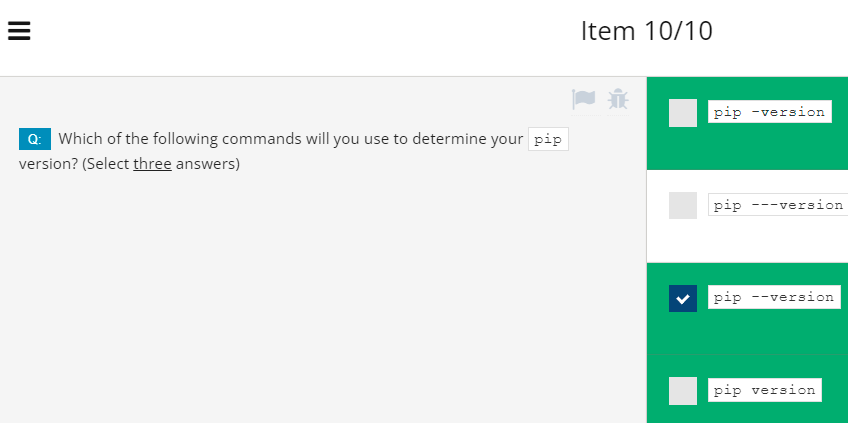




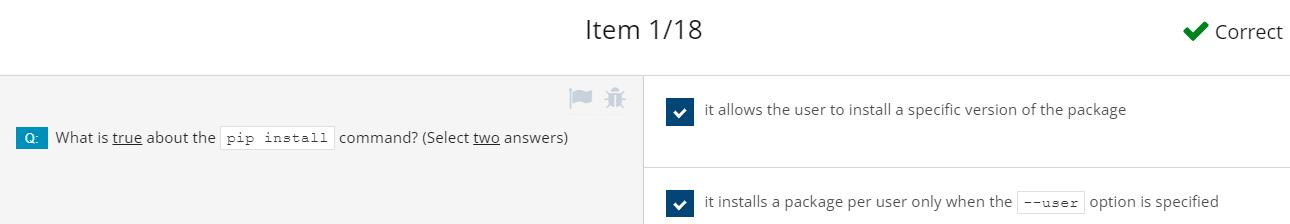


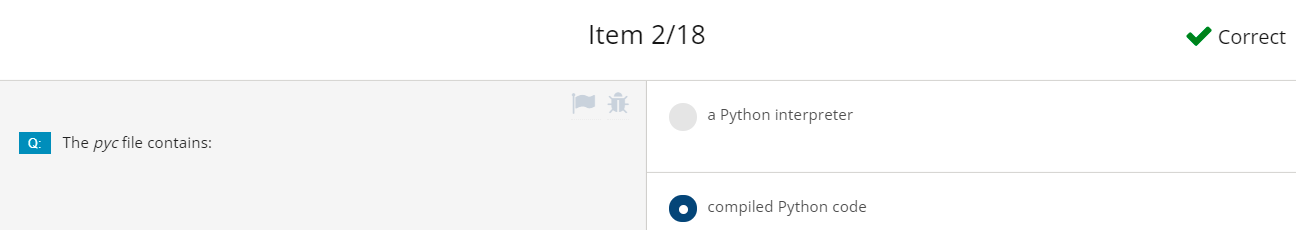


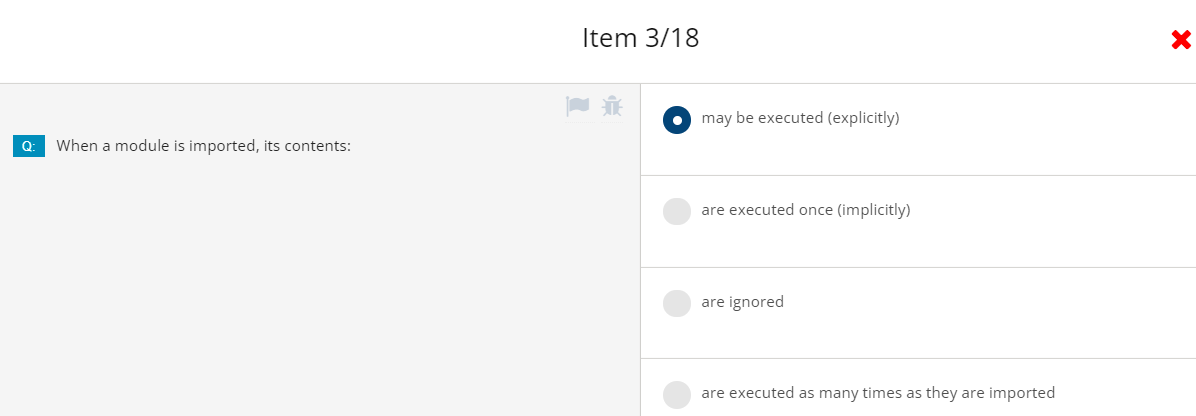




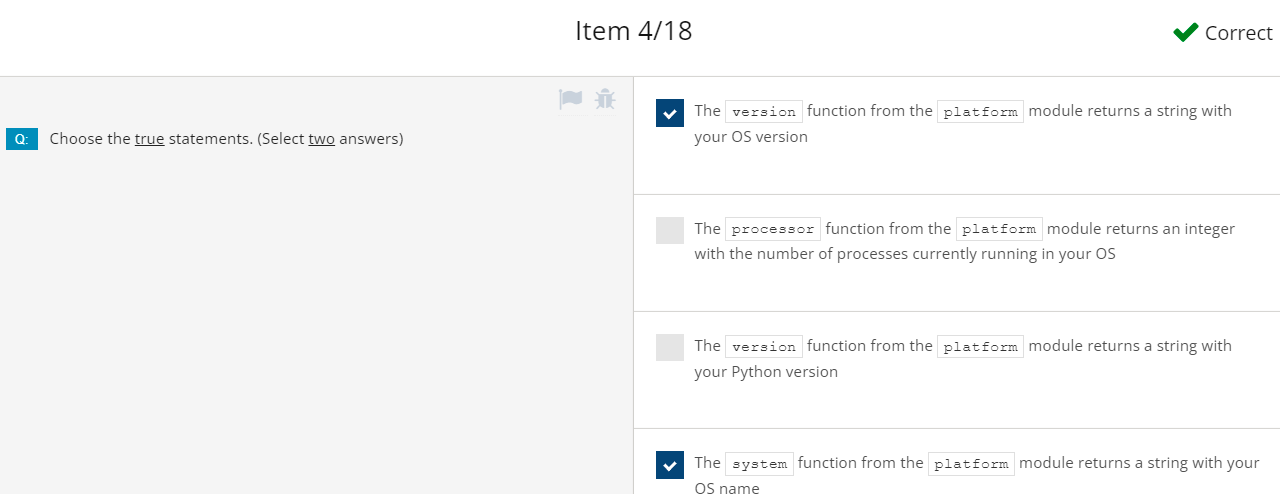
# **Module 1 Test 12/29/21**

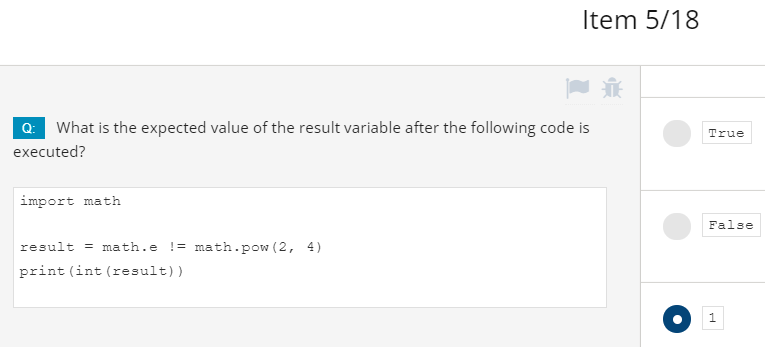


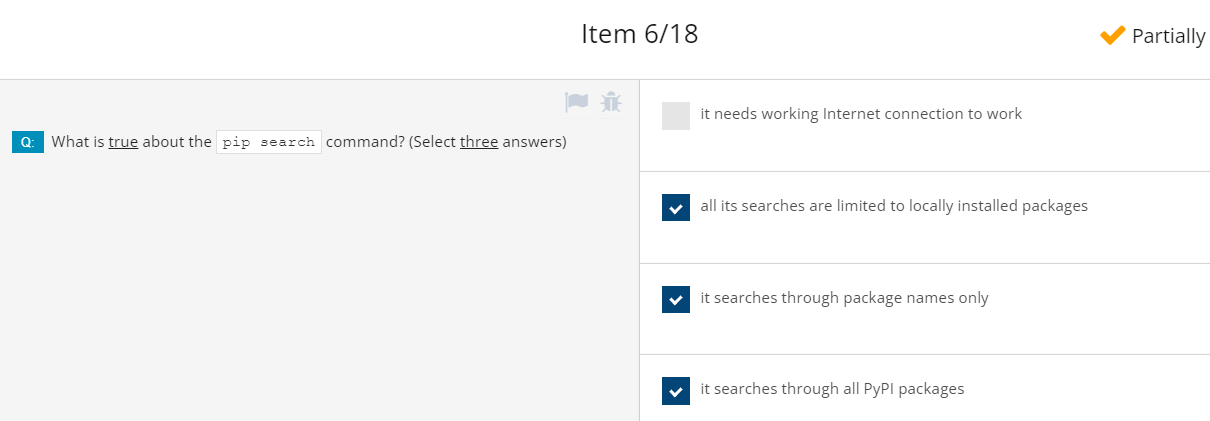




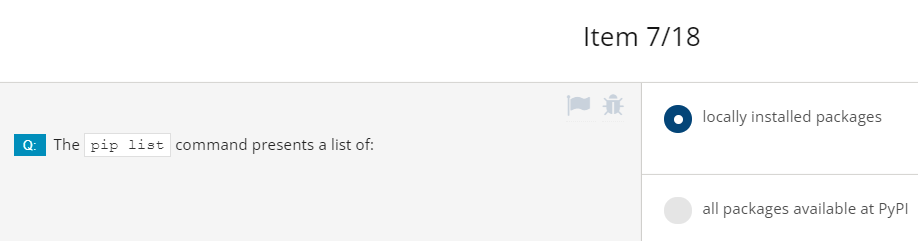
哪个是对的呢？

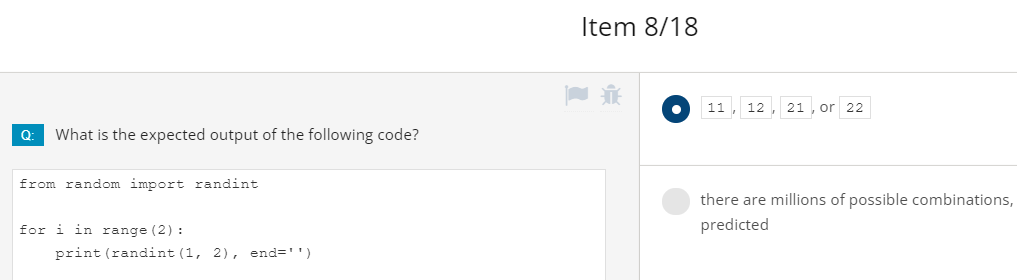


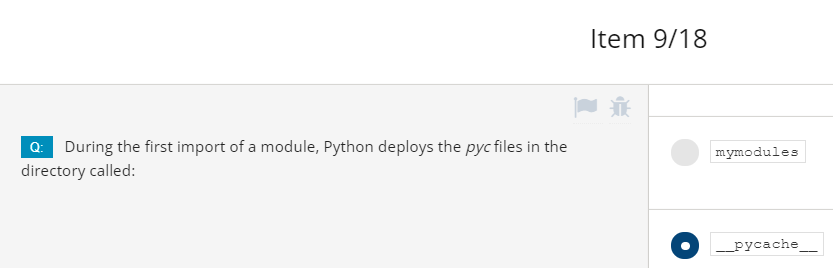


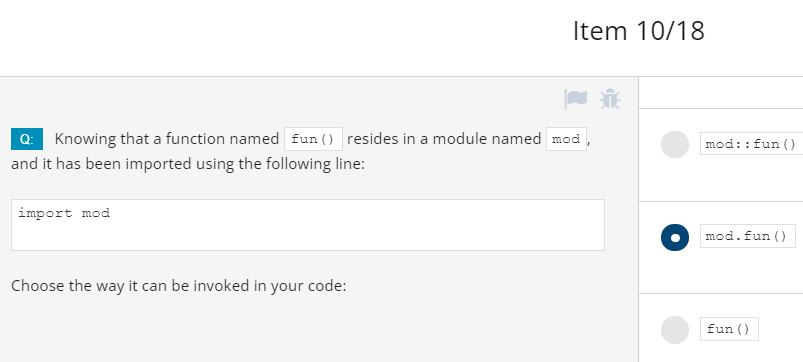


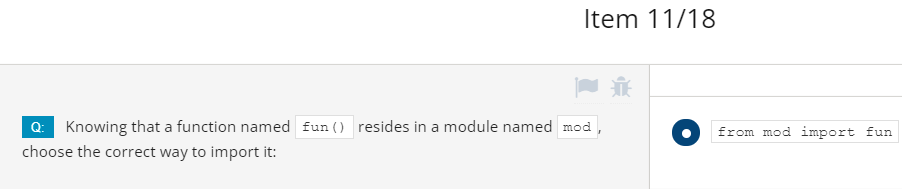
第6题还要研究。

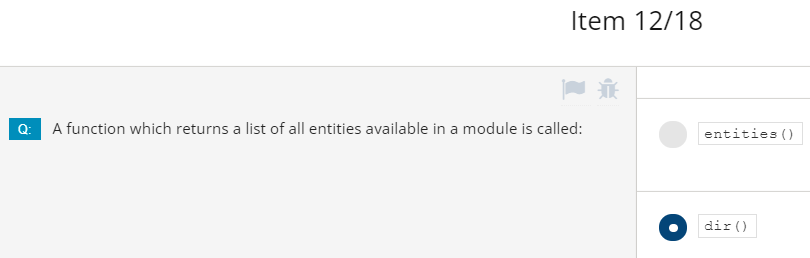


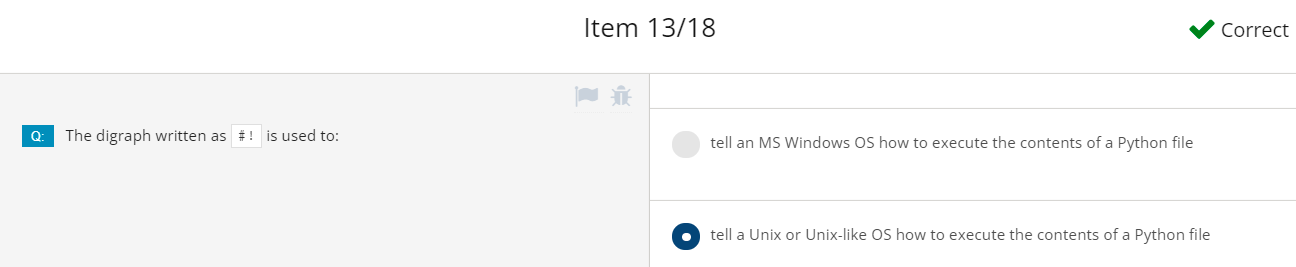


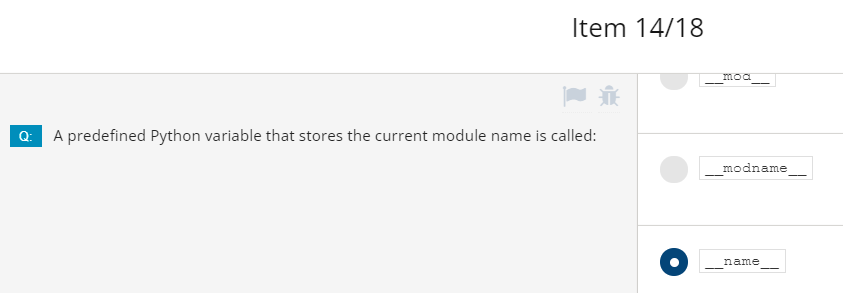


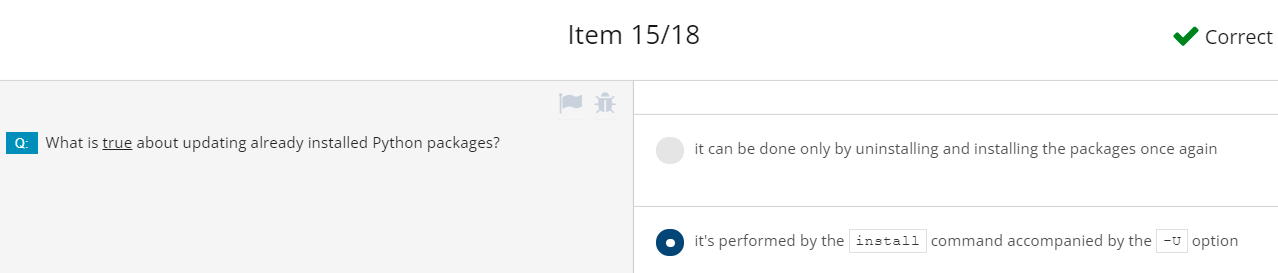


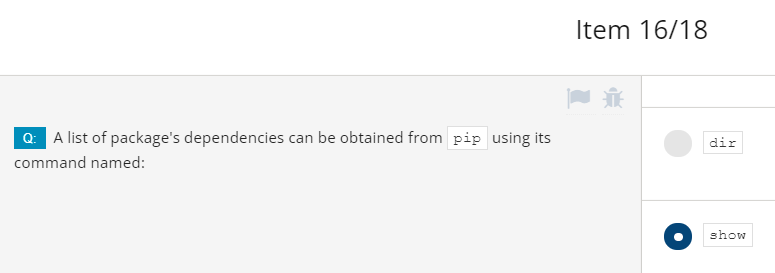


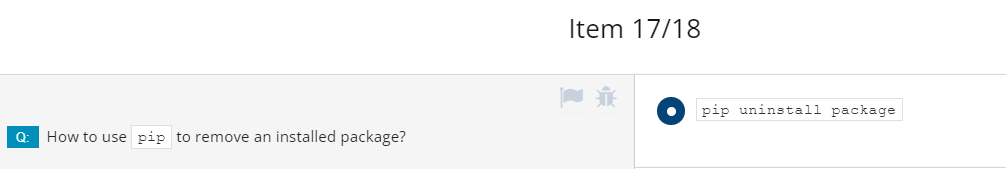


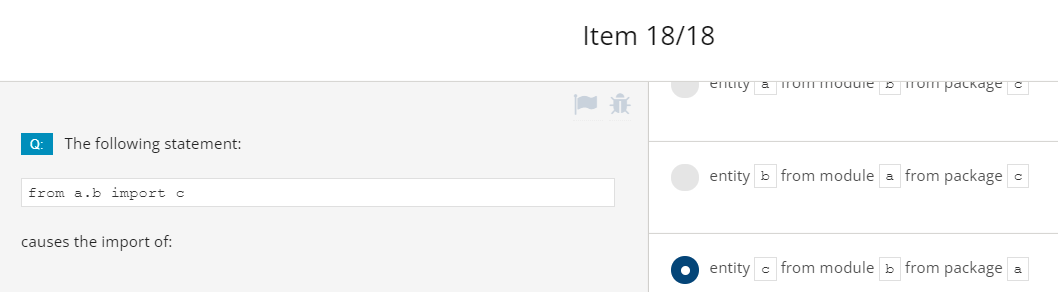












# **MODULE 2: STRINGS, STRING AND LIST METHODS, AND EXCEPTIONS**

# **2.1. CHARACTERS, STRINGS, COMPUTERS**

## **2.1.1.0 Python Essentials 2 - Module 2**

**Python Essentials 2: Module 2 Strings, String and List Methods, Exceptions**

In this module, you will learn about:

* Characters, strings and coding standards;
* Strings vs. lists – similarities and differences;
* Lists methods;
* String methods;
* Python's way of handling runtime errors;
* Controlling the flow of errors using *try* and *except*;
* Hierarchy of exceptions.

## **2.1.1.1 Characters and Strings vs. Computers: ASCII**

**How computers understand single characters**

You've written some interesting programs since you've started this course, but all of them have processed only one kind of data - numbers. As you know (you can see this everywhere around you) lots of computer data are not numbers: first names, last names, addresses, titles, poems, scientific papers, emails, court judgements, love confessions, and much, much more.

All these data must be stored, input, output, searched, and transformed by contemporary computers just like any other data, no matter if they are single characters or multi-volume encyclopedias.

How is it possible?How can you do it in Python? This is what we'll discuss now. Let's start with how computers understand single characters.

**Computers store characters as numbers**. Every character used by a computer corresponds to a unique number, and vice versa. This assignment must include more characters than you might expect. Many of them are invisible to humans, but essential to computers. Some of these characters are called **whitespaces**, while others are named **control characters**, because their purpose is to control input/output devices.

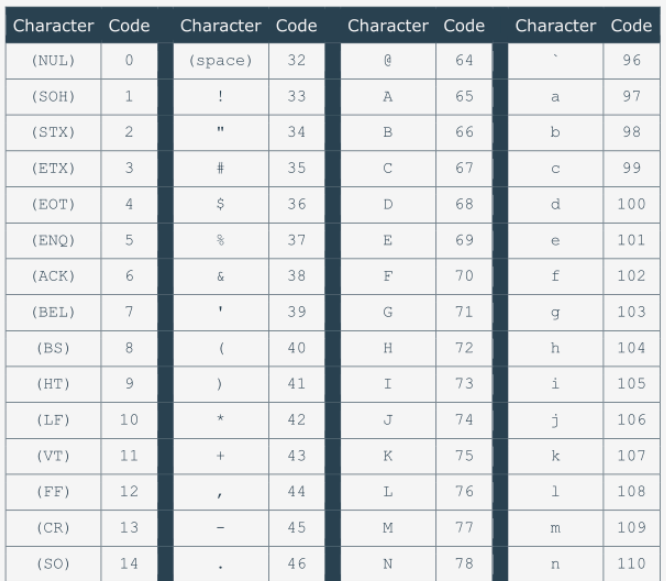
An example of a whitespace that is completely invisible to the naked eye is a special code, or a pair of codes (different operating systems may treat this issue differently), which are used to mark the ends of the lines inside text files.

People do not see this sign (or these signs), but are able to observe the effect of their application where the lines are broken.

We can create virtually any number of character-number assignments, but life in a world in which every type of computer uses a different character encoding would not be very convenient. This system has led to a need to introduce a universal and widely accepted standard implemented by (almost) all computers and operating systems all over the world.

The one named ASCII (short for American Standard Code for Information Interchange) is the most widely used, and you can assume that nearly all modern devices (like computers, printers, mobile phones, tablets, etc.) use that code.

The code provides space for 256 different characters, but we are interested only in the first 128. If you want to see how the code is constructed, look at the table below. Click the table to enlarge it. Look at it carefully - there are some interesting facts. Look at the code of the most common character - the space. This is 32.



Now check the code of the lower-case letter a. This is 97. And now find the upper-case A. Its code is 65. Now work out the difference between the code of a and A. It is equal to 32. That's the code of a space. Interesting, isn't it?

Also note that the letters are arranged in the same order as in the Latin alphabet.

## **2.1.1.2 Characters and Strings vs. Computers: I18N**

Of course, the Latin alphabet is not sufficient for the whole of mankind. Users of that alphabet are in the minority. It was necessary to come up with something more flexible and capacious than ASCII, something able to make all the software in the world amenable to **internationalization**, because different languages use completely different alphabets, and sometimes these alphabets are not as simple as the Latin one.

The word internationalization is commonly shortened to **I18N**.

Why? Look carefully - there is an I at the front of the word, next there are 18 different letters, and an N at the end. Despite the slightly humorous origin, the term is officially used in many documents and standards.

The **software I18N** is a standard in present times. Each program has to be written in a way that enables it to be used all around the world, among different cultures, languages and alphabets.

**A classic form of ASCII code uses eight bits for each sign**. Eight bits mean 256 different characters. The first 128 are used for the standard Latin alphabet (both upper-case and lower-case characters). Is it possible to push all the other national characters used around the world into the remaining 128 locations? No. It isn't.

**Code points and code pages**

We need a new term now: a **code point**. A code point is **a number which makes a character**. For example, 32 is a code point which makes a space in ASCII encoding. We can say that standard ASCII code consists of 128 code points. As standard ASCII occupies 128 out of 256 possible code points, you can only make use of the remaining 128. It's not enough for all possible languages, but it may be sufficient for one language, or for a small group of similar languages.

Can you **set the higher half of the code points differently for different languages?** Yes, you can. Such a concept is called a code page.

A code page is **a standard for using the upper 128 code points to store specific national characters**. For example, there are different code pages for Western Europe and Eastern Europe and Greek alphabets, Arabic and Hebrew languages, and so on.

This means that the one and same code point can make different characters when used in different code pages. For example, the code point 200 makes Č (a letter used by some Slavic languages) when utilized by the ISO/IEC 8859-2 code page, and makes Ш (a Cyrillic letter) when used by the ISO/IEC 8859-5 code page.

In consequence, to determine the meaning of a specific code point, you have to know the target code page. In other words, the code points derived from the code page concept are ambiguous.

## **2.1.1.3 Characters and Strings vs. Computers: Unicode, UCS-4,UTF-8**

**Unicode**

Code pages helped the computer industry to solve I18N issues for some time, but it soon turned out that they would not be a permanent solution. The concept that solved the problem in the long term was **Unicode**.

**Unicode assigns unique (unambiguous) characters (letters, hyphens, ideograms, etc.) to more than a million code points**. The first 128 Unicode code points are identical to ASCII, and the first 256 Unicode code points are identical to the ISO/IEC 8859-1 code page (a code page designed for western European languages).

**UCS-4**

The Unicode standard says nothing about how to code and store the characters in the memory and files. It only names all available characters and assigns them to planes (a group of characters of similar origin, application, or nature).

There is more than one standard describing the techniques used to implement Unicode in actual computers and computer storage systems. The most general of them is **UCS-4**. The name comes from **Universal Character Set**.

**UCS-4 uses 32 bits (four bytes) to store each character**, and the code is just the Unicode code points' unique number. A file containing UCS-4 encoded text may start with a BOM (byte order mark), an unprintable combination of bits announcing the nature of the file's contents. Some utilities may require it.

As you can see, UCS-4 is a rather wasteful standard - it increases a text's size by four times compared to standard ASCII. Fortunately, there are smarter forms of encoding Unicode texts.

**UTF-8**

One of the most commonly used is **UTF-8**. The name is derived from **Unicode Transformation Format**. The concept is very smart. **UTF-8 uses as many bits for each of the code points as it really needs to represent them.** For example:

* all Latin characters (and all standard ASCII characters) occupy eight bits;
* non-Latin characters occupy 16 bits;
* CJK (China-Japan-Korea) ideographs occupy 24 bits.

Due to features of the method used by UTF-8 to store the code points, there is no need to use the BOM, but some of the tools look for it when reading the file, and many editors set it up during the save.

Python 3 fully supports Unicode and UTF-8:

* you can use Unicode/UTF-8 encoded characters to name variables and other entities;
* you can use them during all input and output.

This means that Python3 is completely I18Ned.

## **2.1.1.4 SECTION SUMMARY**

**Key takeaways**

1. Computers store characters as numbers. There is more than one possible way of encoding characters, but only some of them gained worldwide popularity and are commonly used in IT: these are **ASCII** (used mainly to encode the Latin alphabet and some of its derivates) and **UNICODE** (able to encode virtually all alphabets being used by humans).

2. A number corresponding to a particular character is called a **codepoint**.

3. UNICODE uses different ways of encoding when it comes to storing the characters using files or computer memory: two of them are **UCS-4** and **UTF-8** (the latter is the most common as it wastes less memory space).

**Exercise 1:** What is BOM?

Check: BOM (Byte Order Mark) is a special combination of bits announcing encoding used by a file's content (eg. UCS-4 or UTF-B).

**Exercise 2:** Is Python 3 I18Ned?

Check: Yes, it's completely internationalized - we can use UNICODE characters inside our code, read them from input and send to output.

# **2.2. THE NATURE OF STRINGS IN PYTHON**

## **2.2.1.1 The nature of strings in Python: Strings - a brief review**

Let's do a brief review of the nature of Python's strings. First of all, Python's strings (or simply strings, as we're not going to discuss any other language's strings) are **immutable sequences**. It's very important to note this, because it means that you should expect some familiar behavior from them. Let's analyze the code in the editor to understand what we're talking about:

# Example 1

word = 'by'

print(len(word))

# Example 2

empty = ''

print(len(empty))

# Example 3

i\_am = 'I\'m'

print(len(i\_am))

* Take a look at Example 1. The len() function used for strings returns a number of characters contained by the arguments. The snippet outputs 2.
* Any string can be empty. Its length is 0 then - just like in Example 2.
* Don't forget that a backslash (\) used as an escape character is not included in the string's total length. The code in Example 3, therefore, outputs 3.

## **2.2.1.2 The nature of strings in Python: Multiline strings**

Now is a very good moment to show you another way of specifying strings inside the Python source code. Note that the syntax you already know won't let you use a string occupying more than one line of text. For this reason, the code here is erroneous:

multiline = 'Line #1

Line #2'

print(len(multiline))

Fortunately, for these kinds of strings, Python offers separate, convenient, and simple syntax. Look at the code in the editor. This is what it looks like.

multiline = '''Line #1

Line #2'''

print(len(multiline))

As you can see, the string starts with **three apostrophes**, not one. The same tripled apostrophe is used to terminate it. The number of text lines put inside such a string is arbitrary. The snippet outputs 15.

Count the characters carefully. Is this result correct or not? It looks okay at first glance, but when you count the characters, it doesn't.

Line #1 contains seven characters. Two such lines comprise 14 characters. Did we lose a character? Where? How? No, we didn't.

The missing character is simply invisible - it's a **whitespace**. It's located between the two text lines. It's denoted as: \n.

Do you remember? It's a special (control) character used to **force a line feed** (hence its name: LF). You can't see it, but it counts.

The multiline strings can be delimited by **triple quotes**, too, just like here:

multiline = """Line #1

Line #2"""

print(len(multiline))

Choose the method that is more comfortable for you. Both work the same.

## **2.2.1.3 The nature of strings in Python:** Operations on strings

Like other kinds of data, strings have their own set of permissible operations, although they're rather limited compared to numbers. In general, strings can be:

* **concatenated** (joined)
* **replicated**.

The first operation is performed by the + operator (note: it's not an addition) while the second by the \* operator (note again: it's not a multiplication).

The ability to use the same operator against completely different kinds of data (like numbers vs. strings) is called **overloading** (as such an operator is overloaded with different duties). Analyze the example:

str1 = 'a'

str2 = 'b'

print(str1 + str2)

print(str2 + str1)

print(5 \* 'a')

print('b' \* 4)

Output:

ab

ba

aaaaa

bbbb

* The + operator used against two or more strings produces a new string containing all the characters from its arguments (note: the order matters - this overloaded +, in contrast to its numerical version, is not commutative)
* the \* operator needs a string and a number as arguments; here, the order doesn't matter - you can put the number before the string, or vice versa, the result will be the same - a new string created by the nth replication of the argument's string.

Note: shortcut variants of the above operators are also applicable (+= and \*=).

## **2.2.1.4 The nature of strings in Python:** Operations on strings: ord()

If you want **to know a specific character's ASCII/UNICODE code point value**, you can use a function named ord() (as in ordinal). The function needs a **one-character string as its argument** - breaching this requirement causes a *TypeError* exception, and returns a number representing the argument's code point.

Look at the code in the editor, and run it.

# Demonstrating the ord() function.

char\_1 = 'a'

char\_2 = ' ' # space

print(ord(char\_1))

print(ord(char\_2))

The snippet outputs:

97

32

Now assign different values to char\_1 and char\_2, e.g., α (Greek alpha), and ę (a letter in the Polish alphabet); then run the code and see what result it outputs. Carry out your own experiments.

## **2.2.1.5 The nature of strings in Python:** Operations on strings: chr()

If you know the code point (number) and want to get the corresponding character, you can use a function named chr(). The function **takes a code point and returns its character**. Invoking it with an invalid argument (e.g., a negative or invalid code point) causes ValueError or TypeError exceptions. Run the code in the editor.

# Demonstrating the chr() function.

print(chr(97))

print(chr(945))

output

a

α

Note:

chr(ord(x)) == x

ord(chr(x)) == x

Again, run your own experiments.

## **2.2.1.6 The nature of strings in Python:** Strings as sequences: indexing

We told you before that **Python strings are sequences**. It's time to show you what that actually means. Strings aren't lists, but **you can treat them like lists in many particular cases**. For example, if you want to access any of a string's characters, you can do it using indexing, just like in the example below.

# Indexing strings.

the\_string = 'silly walks'

for ix in range(len(the\_string)):

print(the\_string[ix], end=' ')

print()

Be careful - don't try to pass a string's boundaries - it will cause an exception. The example output is: s i l l y w a l k s

By the way, negative indices behave as expected, too. Check this yourself.

**Strings as sequences: iterating**

Iterating through the strings works, too. Look at the example below:

# Iterating through a string.

the\_string = 'silly walks'

for character in the\_string:

print(character, end=' ')

print()

The output is the same as previously. Check.

## **2.2.1.7 The nature of strings in Python:** Slices

Moreover, everything you know about **slices** is still usable. We've gathered some examples showing how slices work in the string world. Look at the code in the editor, analyze it, and run it.

# Slices

alpha = "abdefg"

print(alpha[1:3])

print(alpha[3:])

print(alpha[:3])

print(alpha[3:-2])

print(alpha[-3:4])

print(alpha[::2])

print(alpha[1::2])

output

bd

efg

abd

e

e

adf

beg

You won't see anything new in the example, but we want you to be sure that you can explain all the lines of the code.

*Note: The syntax of a Slice operator using double colon is [****Start*** *:* ***Stop*** *:* ***Steps****].* ***Start*** *(Indicates the number from where the slicing will start),* ***Stop****(Indicates the number where the slicing will stop) and* ***Steps****(Indicates the number of jumps interpreter will take to slice the string) are the three flags and all these flags are integer values.(<https://www.askpython.com/python/examples/colon-in-python>)*

## **2.2.1.8 The nature of strings in Python:** The in and not in operators

**The in operator**

The in operator shouldn't surprise you when applied to strings - it simply **checks if its left argument (a string) can be found anywhere within the right argument (another string).** The result of the check is simply True or False. Look at the example program below. This is how the in operator works:

al = "abcdefghijklmnopqrstuvwxyz"

print("f" in al)

print("F" in al)

print("1" in al)

print("ghi" in al)

print("Xyz" in al)

The example output is:

True

False

False

True

False

**The not in operator**

The not in operator is also applicable here. This is how it works:

al = "abcdefghijklmnopqrstuvwxyz"

print("f" not in a)

print("F" not in a)

print("1" not in al)

print("ghi" not in al)

print("Xyz" not in al)

The example output is:

False

True

True

False

True

## **2.2.1.9 The nature of strings in Python:** Python strings are immutable

We've also told you that Python's **strings are immutable**. This is a very important feature. What does it mean? This primarily means that the similarity of strings and lists is limited. Not everything you can do with a list may be done with a string. The first important difference **doesn't allow you to use the *del* instruction to remove anything from a string.** The example here won't work:

alphabet = "abcdefghijklmnopqrstuvwxyz"

del alphabet[0]

The only thing you can do with del and a string is to **remove the string as a whole**. Try to do it.

al = "abcdefghijklmnopqrstuvwxyz"

# Write test code here.

del al

print(al)

Traceback (most recent call last):

File "main.py", line 5, in <module>

print(alphabet)

NameError: name 'alphabet' is not defined

Python strings **don't have the append() method** - you cannot expand them in any way. The example below is erroneous:

alphabet = "abcdefghijklmnopqrstuvwxyz"

alphabet.append("A")

with the absence of the append() method, the insert() method is illegal, too:

alphabet = "abcdefghijklmnopqrstuvwxyz"

alphabet.insert(0, "A")

## **2.2.1.10 The nature of strings in Python:** Operations on strings: continued

Don't think that a string's immutability limits your ability to operate with strings. The only consequence is that you have to remember about it, and implement your code in a slightly different way - look at the example code in the editor.

alphabet = "bcdefghijklmnopqrstuvwxy"

alphabet = "a" + alphabet

alphabet = alphabet + "z"

print(alphabet)

This form of code is fully acceptable, will work without bending Python's rules, and will bring the full Latin alphabet to your screen:

abcdefghijklmnopqrstuvwxyz

You may want to ask **if creating a new copy of a string each time you modify its contents worsens the effectiveness of the code.** Yes, it does. A bit. It's not a problem at all, though.

## **2.2.1.11 The nature of strings in Python:** Operations on strings: min()

Now that you understand that strings are sequences, we can show you some less obvious sequence capabilities. We'll present them using strings, but don't forget that lists can adopt the same tricks, too. Let's start with a function named *min*().

The function **finds the minimum element of the sequence passed as an argument.** There is one condition - the sequence (string, list, it doesn't matter) **cannot be empty**, or else you'll get a ValueError exception.

# Demonstrating min() - Example 1:

print(min("aAbByYzZ"))

# Demonstrating min() - Examples 2 & 3:

t = 'The Knights Who Say "Ni!"'

print('[' + min(t) + ']')

t = [0, 1, 2]

print(min(t))

The **Example** 1 program outputs: A. Note: It's an upper-case A. Why? Recall the ASCII table - which letters occupy first locations? We've prepared two more examples to analyze: **Examples** 2 & 3. As you can see, they present more than just strings. The expected output looks as follows:

[ ]

0

Note: square brackets is to prevent the space from being overlooked on your screen.

## **2.2.1.12 The nature of strings in Python:** Operations on strings: max()

Similarly, a function named max() finds the maximum element of the sequence. Look at Example 1 in the editor.

# Demonstrating max() - Example 1:

print(max("aAbByYzZ"))

# Demonstrating max() - Examples 2 & 3:

t = 'The Knights Who Say "Ni!"'

print('[' + max(t) + ']')

t = [0, 1, 2]

print(max(t))

The example program outputs: z. Note: It's a lower-case z.

Now let's see the max() function applied to the same data as previously. Look at Examples 2 & 3 in the editor.The expected output is:

[y]

2

## **2.2.1.13 The nature of strings in Python:** the index() method

The index() method (it's a method, not a function) **searches the sequence from the beginning, in order to find the first element of the value specified in its argument.**

Note: the element searched for must occur in the sequence - **its absence will cause a ValueError exception.**

The method returns **the index of the first occurrence of the argument** (which means that the lowest possible result is 0, while the highest is the length of argument decremented by 1).

# Demonstrate the index() method:

print("aAbByYzZaA".index("b"))

print("aAbByYzZaA".index("Z"))

print("aAbByYzZaA".index("A"))

outputs:

2

7

1

## **2.2.1.14 The nature of strings in Python:** Operations on strings: the list() function

The list() function **takes its argument (a string) and creates a new list containing all the string's characters, one per list element.**

Note: it's not strictly a string function - list() is able to create a new list from many other entities (e.g., from tuples and dictionaries). Take a look at the code example in the editor.

# Demonstrating the list() function:

print(list("abcabc"))

#Demonstrating the count() method:

print("abcabc".count("b"))

print('abcabc'.count("d"))

The example outputs:

['a', 'b', 'c', 'a', 'b', 'c']

2

0

**Operations on strings: the count() method**

The count() method **counts all occurrences of the element inside the sequence.** The absence of such elements doesn't cause any problems. Look at the second example in the editor. Can you guess its output?

Moreover, Python strings have a significant number of methods intended exclusively for processing characters. Don't expect them to work with any other collections. The complete list of is presented here: <https://docs.python.org/3.4/library/stdtypes.html#string-methods.> We're going to show you the ones we consider the most useful.

## **2.2.1.15 SECTION SUMMARY**

**Key takeaways**

1. Python strings are **immutable sequences** and can be indexed, sliced, and iterated like any other sequence, as well as being subject to the *in* and *not in* operators. There are two kinds of strings in Python:

* **one-line** strings, which cannot cross line boundaries – we denote them using either apostrophes ('string') or quotes ("string")
* **multi-line** strings, which occupy more than one line of source code, delimited by trigraphs:

'''

string

'''

or

"""

string

"""

2. The length of a string is determined by the len() function. The escape character (\) is not counted. For example: print(len("\n\n")) outputs 2.

3. Strings can be **concatenated** using the + operator, and **replicated** using the \* operator. For example:

asterisk = '\*'

plus = "+"

decoration = (asterisk + plus) \* 4 + asterisk

print(decoration)

outputs \*+\*+\*+\*+\*.

4. The pair of functions chr() and ord() can be used to create a character using its codepoint, and to determine a codepoint corresponding to a character. Both of the following expressions are always true:

chr(ord(character)) == character

ord(chr(codepoint)) == codepoint

5. Some other functions that can be applied to strings are:

* list() – create a list consisting of all the string's characters;
* max() – finds the character with the maximal codepoint;
* min() – finds the character with the minimal codepoint.

6.The method named index() finds the index of a given substring inside the string.

**Exercise 1:** What is the length of the following string assuming there is no whitespaces between the quotes?

"""

"""

Check: 1

**Exercise 2:** What is the expected output of the following code?

s = 'yesteryears'

the\_list = list(s)

print(the\_list[3:6])

Check: ['t', 'e', 'r']

**Exercise 3:** What is the expected output of the following code?

for ch in "abc":

print(chr(ord(ch) + 1), end='')

Check: bcd

# **2.3. STRING METHODS**

## **2.3.1.1 String methods:** The capitalize() method

Let's go through some standard Python string methods. We're going to go through them in alphabetical order - to be honest, any order has as many disadvantages as advantages, so the choice may as well be random.

The capitalize() method does exactly what it says - **it creates a new string filled with characters taken from the source string**, but it tries to modify them in the following way:

* **if the first character inside the string is a letter** (note: the first character is an element with an index equal to 0, not just the first visible character), **it will be converted to upper-case**;
* **all remaining letters from the string will be converted to lower-case.**

Don't forget that:

* the original string (from which the method is invoked) is not changed in any way (a string's immutability must be obeyed without reservation)
* the modified (capitalized in this case) string is returned as a result - if you don't use it in any way (assign it to a variable, or pass it to a function/method) it will disappear without a trace.

Note: methods don't have to be invoked from within variables only. They can be invoked directly from within string literals. We're going to use that convention regularly - it will simplify the examples, as the most important aspects will not disappear among unnecessary assignments. Take a look at the example in the editor.

# Demonstrating the capitalize() method:

print('aBcD'.capitalize())

This is what it prints: Abcd

Try some more advanced examples and test their output:

print("Alpha".capitalize())

print('ALPHA'.capitalize())

print(' Alpha'.capitalize())

print('123'.capitalize())

print("αβγδ".capitalize())

Output:

Alpha

Alpha

alpha

123

Αβγδ

## **2.3.1.2 String methods:** The center() method

The one-parameter variant of the center() method makes a copy of the original string, trying to center it inside a field of a specified width. The centering is actually done by **adding some spaces before and after the string**. Don't expect this method to demonstrate any sophisticated skills. It's rather simple.

The example in the editor uses brackets to clearly show you where the centered string actually begins and terminates.

# Demonstrating the center() method:

print('[' + 'alpha'.center(10) + ']')

Its output looks as follows: [ alpha ]

If the target field's length is too small to fit the string, the original string is returned. You can see the center() method in more examples here:

print('[' + 'Beta'.center(2) + ']')

print('[' + 'Beta'.center(4) + ']')

print('[' + 'Beta'.center(6) + ']')

Output:

[Beta]

[Beta]

[ Beta ]

**The two-parameter variant of center() makes use of the character from the second argument, instead of a space**. Analyze the example below:

print('[' + 'gamma'.center(20, '\*') + ']')

This is why the output now looks like this: [\*\*\*\*\*\*\*gamma\*\*\*\*\*\*\*\*]

## **2.3.1.3 String methods:** The endswith() method

The endswith() method **checks if the given string ends with the specified argument and returns True or False**, depending on the check result.

Note: the substring must adhere to the string's last character - it cannot just be located somewhere near the end of the string. Look at our example, analyze it, and run it.

# Demonstrating the endswith() method:

if "epsilon".endswith("on"):

print("yes")

else:

print("no")

It outputs: yes. You should now be able to predict the output of the snippet below:

t = "zeta"

print(t.endswith("a"))

print(t.endswith("A"))

print(t.endswith("et"))

print(t.endswith("eta"))

Output:

True

False

False

True

## **2.3.1.4 String methods:** The find() method

The find() method is similar to index(), which you already know - **it looks for a substring and returns the index of first occurrence of this substring**, but:

* it's safer - **it doesn't generate an error for an argument containing a non-existent substring** (it returns -1 then)
* it **works with strings only** - don't try to apply it to any other sequence.

Look at the code in the editor. This is how you can use it.

# Demonstrating the find() method:

print("Eta".find("ta"))

print("Eta".find("mma"))

The example prints:

1

-1

Note: don't use find() if you only want to check if a single character occurs within a string - the in operator will be significantly faster. Here is another example:

t = 'theta'

print(t.find('eta'))

print(t.find('et'))

print(t.find('the'))

print(t.find('ha'))

Output:

2

2

0

-1

If you want to perform the find, not from the string's beginning, but **from any position**, you can use a **two-parameter variant** of the find() method. Look at the example:

print('kappa'.find('a', 2)) #output: 4

The 2nd argument **specifies the index at which the search will be started** (it doesn't have to fit inside the string). Among the two a letters, only the 2nd will be found.

You can use the find() method to search for all the substring's occurrences, like here:

the\_text = """A variation of the ordinary lorem ipsum text has been used in type setting since the 1960s or earlier, when it was popularized by advertisements

for Letraset transfer sheets. It was introduced to the Information Age in the mid-1980s by the Aldus Corporation, which employed it in graphics and word-processing templatesfor its desktop publishing program PageMaker """

fnd = the\_text.find('the')

while fnd != -1:

print(fnd)

fnd = the\_text.find('the', fnd + 1)

The code prints the indices of all occurrences of the article *the*, and its output:

15

80

198

221

238

There is also a **three-parameter mutation of the find() method** - the third argument **points to the first index which won't be taken into consideration during the search** (it's actually the upper limit of the search). Look at our example below:

print('kappa'.find('a', 1, 4)) #output: 1

print('kappa'.find('a', 2, 4)) #output: -1

The second argument specifies the index at which the search will be started (it doesn't have to fit inside the string). (a cannot be found within the given search boundaries in the second print().

## **2.3.1.5 String methods:** The isalnum() method

The parameterless method named isalnum() checks if the string contains only digits or alphabetical characters (letters), and returns True or False according to the result.

Look at the example in the editor and run it.

# Demonstrate isalnum() method:

print('lambda30'.isalnum())

print('lambda'.isalnum())

print('30'.isalnum())

print('@'.isalnum())

print('lambda\_30'.isalnum())

print(''.isalnum())

The example output is:

True

True

True

False

False

False

Note: any string element that is not a digit or a letter causes the method to return False. An empty string does, too.

Three more intriguing examples are here:

t = 'Six lambdas'

print(t.isalnum()) #False

t = 'ΑβΓδ'

print(t.isalnum())#True

t = '20E1'

print(t.isalnum())#True

Hint: the cause of the first result is a space - it's neither a digit nor a letter.

## **2.3.1.6 String methods:** The isalpha() and **isdigit()** method

The isalpha() method is more specialized - it's interested in **letters only**. Look at Example 1 - its output is:

# Example 1: Demonstrating the isapha() method:

print("Moooo".isalpha()) # True

print('Mu40'.isalpha()) # False

**The isdigit() method**

In turn, the isdigit() method looks at digits only - anything else produces False as the result. Look at Example 2 - its output is:

# Example 2: Demonstrating the isdigit() method:

print('2018'.isdigit()) # True

print("Year2019".isdigit()) # False

Carry out more experiments.

## **2.3.1.7 String methods:** The **islower**(), **isspace(),** **isupper()** method

**The islower() method**

The islower() method is a fussy variant of isalpha() - it accepts lower-case letters only. Look at Example 1 in the editor - it outputs:

# Example 1: Demonstrating the islower() method:

print("Moooo".islower()) #False

print('moooo'.islower()) #True

**The isspace() method**

The isspace() method identifies **whitespaces only** - it disregards any other character (the result is False then). Look at Example 2 in the editor - the output is:

# Example 2: Demonstrating the isspace() method:

print(' \n '.isspace()) #True

print(" ".isspace()) #True

print("mooo mooo mooo".isspace()) #False

**The isupper() method**

The isupper() method is the upper-case version of islower() - it concentrates on **upper-case letters only**. Again, Look at the code in the editor - Example 3 produces the following output:

# Example 3: Demonstrating the isupper() method:

print("Moooo".isupper()) # False

print('moooo'.isupper()) # False

print('MOOOO'.isupper()) # True

## **2.3.1.8 String methods:** The join() method

The join() method is rather complicated, so let’s guide step by step thorough it:

* as its name suggests, the method **performs a join** - it expects one argument as a list; it must be assured that all the list's elements are strings - the method will raise a TypeError exception otherwise;
* all the list's elements will be **joined into one string** but...
* ...the string from which the method has been invoked is **used as a separator**, put among the strings;
* the newly created string is returned as a result.

Take a look at the example in the editor. Let's analyze it:

# Demonstrating the join() method:

print(",".join(["omicron", "pi", "rho"]))

* the join() method is invoked from within a string containing a comma (the string can be arbitrarily long, or it can be empty)
* the join's argument is a list containing three strings;
* the method returns a new string.

Here it is: omicron,pi,rho

## **2.3.1.9 String methods:** The lower() method

The lower() method **makes a copy of a source string, replaces all upper-case letters with their lower-case counterparts**, and returns the string as the result. Again, the source string remains untouched. If the string doesn't contain any upper-case characters, the method returns the original string.

Note: The lower() method doesn't take any parameters.

# Demonstrating the lower() method:

print("SiGmA=60".lower())

The example in the editor outputs:

sigma=60

## **2.3.1.10 String methods:** The lstrip() method

The parameterless lstrip() method **returns a newly created string formed from the original one by removing all leading whitespaces.** Analyze the example code in the editor. The brackets are not a part of the result - they only show the result's boundaries. The example outputs:

# Demonstrating the lstrip() method:

print("[" + " tau ".lstrip() + "]")

output

[tau ]

The **one-parameter** lstrip() method does the same as its parameterless version, but removes all characters enlisted in its argument (a string), not just whitespaces:

print("www.cisco.com".lstrip("w."))

Brackets aren't needed here, as the output looks as follows: cisco.com

Can you guess the output of the snippet below? Think carefully. Run the code and check your predictions.

print("pythoninstitute.org".lstrip(".org")) #output:pythoninstitute.org

Surprised? **Leading** characters, leading whitespaces. Again, experiment with your own examples.

## **2.3.1.11 String methods:** The replace() method 12/30/21

**The two-parameter replace() method returns a copy of the original string in which all occurrences of the first argument have been replaced by the second argument.** Look at the example code in the editor.

# Demonstrating the replace() method:

print("www.netacad.com".replace("netacad.com", "pythoninstitute.org"))

#www.pythoninstitute.org

print("This is it!".replace("is", "are")) #Thare are it!

print("Apple juice".replace("juice", "")) #Apple

If the second argument is an empty string, **replacing is actually removing** the first argument's string. What will happen if the first argument is an empty string?

**The three-parameter** replace() variant uses the third argument (a number) to limit the number of replacements.Look at the modified example code below:

print("This is it!".replace("is", "are", 1)) # Thare is it!

print("This is it!".replace("is", "are", 2)) #Thare are it!

## **2.3.1.12 String methods:** The rfind() method

The one-, two-, and three-parameter methods named rfind() do nearly the same things as their counterparts (the ones devoid of the r prefix), but **start their searches from the end of the string**, not the beginning (hence the prefix r, for right).

Take a look at the example code in the editor and try to predict its output. Run the code to check if you were right.

# Demonstrating the rfind() method:

print("tau tau tau".rfind("ta")) #8

print("tau tau tau".rfind("ta", 9)) #-1

print("tau tau tau".rfind("ta", 3, 9)) #4

## **2.3.1.13 String methods:** The rstrip() method

Two variants of the rstrip() method do nearly the same as lstrips, but affect the opposite side of the string. Look at the code example in the editor. Can you guess its output? Run the code to check your guesses.

# Demonstrating the rstrip() method:

print("[" + " upsilon ".rstrip() + "]") # [ upsilon]

print("cisco.com".rstrip(".com")) # cis

## **2.3.1.14 String methods:** The split() method

The split() method **splits the string and builds a list of all detected substrings.** The method **assumes that the substrings are delimited by whitespaces** - the spaces don't take part in the operation, and aren't copied into the resulting list. If the string is empty, the resulting list is empty too. Look at the code in the editor.

# Demonstrating the split() method:

print("phi chi\npsi".split()) #['phi', 'chi', 'psi']

Note: the reverse operation can be performed by the join() method.

## **2.3.1.15 String methods:** The startswith() and **strip()** method

The startswith() method is a mirror reflection of endswith() - it **checks if a given string starts with the specified substring**. Look at the example in the editor:

# Demonstrating the startswith() method:

print("omega".startswith("meg")) #False

print("omega".startswith("om")) #True

**The strip() method**

The strip() method combines the effects caused by rstrip() and lstrip() - it **makes a new string lacking all the leading and trailing whitespaces.** Look at the example :

# Demonstrating the strip() method:

print("[" + " aleph ".strip() + "]") # [aleph]

## **2.3.1.16 String methods:** The swapcase(), **title(), upper()** method

The swapcase() method **makes a new string by swapping the case of all letters within the source string**: lower-case characters become upper-case, and vice versa. All other characters remain untouched.

Look at the first example in the editor. Can you guess the output?

# Demonstrating the swapcase() method:

print("I know that I know nothing.".swapcase())

Output: i KNOW THAT i KNOW NOTHING.

**The title() method**

The title() method performs a somewhat similar function - it **changes every word's first letter to upper-case, turning all other ones to lower-case.** Look at the second example in the editor. Can you guess its output? This is the result:

# Demonstrating the title() method:

print("I know that I know nothing. Part 1.".title())

Output: I Know That I Know Nothing. Part 1.

**The upper() method**

Last but not least, the upper() method **makes a copy of the source string, replaces all lower-case letters with their upper-case counterparts**, and returns the string as the result. Look at the third example in the editor. It outputs:

# Demonstrating the upper() method:

print("I know that I know nothing. Part 2.".upper())

#output I KNOW THAT I KNOW NOTHING. PART 2.

Hoooray! We've made it to the end of this section. Are you surprised with any of the string methods we've discussed so far? Take a couple of minutes to review them, and let's move on to the next part of the course where we'll show you what great things we can do with strings.

## **2.3.1.17 SECTION SUMMARY**

**Key takeaways**

1. Some of the methods offered by strings are:

* capitalize() – changes all string letters to capitals;
* center() – centers the string inside the field of a known length;
* count() – counts the occurrences of a given character;
* join() – joins all items of a tuple/list into one string;
* lower() – converts all the string's letters into lower-case letters;
* lstrip() – removes the white characters from the beginning of the string;
* replace() – replaces a given substring with another;
* rfind() – finds a substring starting from the end of the string;
* rstrip() – removes the trailing white spaces from the end of the string;
* split() – splits the string into a substring using a given delimiter;
* strip() – removes the leading and trailing white spaces;
* swapcase() – swaps the letters' cases (lower to upper and vice versa)
* title() – makes the first letter in each word upper-case;
* upper() – converts all the string's letter into upper-case letters.

2. String content can be determined using the following methods (all of them return Boolean values):

* endswith() – does the string end with a given substring?
* isalnum() – does the string consist only of letters and digits?
* isalpha() – does the string consist only of letters?
* islower() – does the string consists only of lower-case letters?
* isspace() – does the string consists only of white spaces?
* isupper() – does the string consists only of upper-case letters?
* startswith() – does the string begin with a given substring?

**Exercise 1: What is the expected output of the following code?**

for ch in "abc123XYX":

if ch.isupper():

print(ch.lower(), end='')

elif ch.islower():

print(ch.upper(), end='')

else:

print(ch, end='')

Check: ABC123xyz

**Exercise 2: What is the expected output of the following code?**

s1 = 'Where are the snows of yesteryear?'

s2 = s1.split()

print(s2[-2])

Check: of

**Exercise 3: What is the expected output of the following code?**

the\_list = ['Where', 'are', 'the', 'snows?']

s = '\*'.join(the\_list)

print(s)

Check: Where\*are\*the\*snows?

**Exercise 4: What is the expected output of the following code?**

s = 'It is either easy or impossible'

s = s.replace('easy', 'hard').replace('im', '')

print(s)

Check: It is either hard or possible

## **2.3.1.18 Your own split**

**LAB**

Estimated time: 20-25 minutes

Level of difficulty: Medium

**Objectives**

* improving the student's skills in operating with strings;
* using built-in Python string methods.

**Scenario**

You already know how split() works. Now we want you to prove it. Your task is to **write your own function, which behaves almost exactly like the original split() method**, i.e.:

* it should accept exactly one argument - a string;
* it should return a list of words created from the string, divided in the places where the string contains whitespaces;
* if the string is empty, the function should return an empty list;
* its name should be mysplit()

Use the template in the editor. Test your code carefully.

def mysplit(strng):

#

# put your code here

#

A = ""

B = []

for i in strng:

if i != " ":

A += i

elif A != "":

B.append(A)

A = ""

# Append last word

if A != "":

B.append(A)

return(B)

print(mysplit("To be or not to be, that is the question"))

print(mysplit("To be or not to be,that is the question"))

print(mysplit(" "))

print(mysplit(" abc "))

print(mysplit(""))

**Expected output**

['To', 'be', 'or', 'not', 'to', 'be,', 'that', 'is', 'the', 'question']

['To', 'be', 'or', 'not', 'to', 'be,that', 'is', 'the', 'question']

[]

['abc']

[]

# **2.4. STRINGS IN ACTION AND LIST METHODS**

## 2.4.1.1 String in action: Comparing strings

Python's strings **can be compared using the same set of operators** which are in use in relation to numbers. Look at these operators - they can all compare strings, too:

* + ==
  + !=
  + >
  + >=
  + <
  + <=

There is one "but" - the results of such comparisons may sometimes be a bit surprising. Don't forget that Python is not aware (it cannot be in any way) of subtle linguistic issues - it just compares code point values, character by character.

The results you get from such an operation are sometimes astonishing. Let's start with the simplest cases.

Two strings are equal when they consist of the same characters in the same order. By the same fashion, two strings are not equal when they don't consist of the same characters in the same order. Both comparisons give True as a result:

'alpha' == 'alpha'

'alpha' != 'Alpha'

The final relation between strings is determined by **comparing the first different character in both strings** (keep ASCII/UNICODE code points in mind at all times.)

When you compare two strings of different lengths and the shorter one is identical to the longer one's beginning, **the longer string is considered greater**.

Just like here: 'alpha' < 'alphabet'. The relation is True.

String comparison is always case-sensitive (**upper-case letters are taken as lesser than lower-case**). The expression is True: 'beta' > 'Beta'.

## 2.4.1.2 String in action: Comparing strings: continued

Even **if a string contains digits only, it's still not a number.** It's interpreted as-is, like any other regular string, and its (potential) numerical aspect is not taken into consideration in any way. Look at the examples:

'10' == '010'

'10' > '010'

'10' > '8'

'20' < '8'

'20' < '80'

They produce the following results:

False

True

False

True

True

**Comparing strings against numbers is generally a bad idea.**

The only comparisons you can perform with impunity are these symbolized by the == and != operators. The former always gives False, while the latter always produces True. Using any of the remaining comparison operators will raise a TypeError exception.

Let's check it:

'10' == 10

'10' != 10

'10' == 1

'10' != 1

'10' > 10

The results in this case are:

False

True

False

True

TypeError exception

## 2.4.1.3 String in action: Sorting

Comparing is closely related to sorting (or rather, sorting is in fact a very sophisticated case of comparing).

This is a good opportunity to show you two possible ways to **sort lists containing strings**. Such an operation is very common in the real world - any time you see a list of names, goods, titles, or cities, you expect them to be sorted. Let's assume that you want to sort the following list:

greek = ['omega', 'alpha', 'pi', 'gamma']

In general, Python offers two different ways to sort lists. The first is implemented as a **function named** sorted(). The function takes one argument (a list) and returns a new list, filled with the sorted argument's elements. (Note: this description is a bit simplified compared to the actual implementation - we'll discuss it later.) The original list remains untouched.

Look at the code in the editor, and run it. The snippet produces the following output:

# Demonstrating the sorted() function:

first\_greek = ['omega', 'alpha', 'pi', 'gamma']

first\_greek\_2 = sorted(first\_greek)

print(first\_greek) # ['omega', 'alpha', 'pi', 'gamma']

print(first\_greek\_2) # ['alpha', 'gamma', 'omega', 'pi']

The second method affects the list itself - **no new list is created**. Ordering is performed in situ by the method named sort(). The output hasn't changed:

# Demonstrating the sort() method:

second\_greek = ['omega', 'alpha', 'pi', 'gamma']

print(second\_greek) # ['omega', 'alpha', 'pi', 'gamma']

second\_greek.sort()

print(second\_greek) #['alpha', 'gamma', 'omega', 'pi']

If you need an order other than non-descending, you have to convince the function/method to change its default behaviors. We'll discuss it soon.

## 2.4.1.4 String in action: Strings vs. numbers

There are two additional issues that should be discussed here: **how to convert a number (an integer or a float) into a string, and vice versa.** It may be necessary to perform such a transformation. Moreover, it's a routine way to process input/output data. The number-string conversion is simple, as it is always possible. It's done by a function named *str().* Just like here:

itg = 13

flt = 1.3

si = str(itg)

sf = str(flt)

print(si + ' ' + sf)

The code outputs: 13 1.3

The reverse transformation (string-number) is possible when and only when the string represents a valid number. If the condition is not met, expect a ValueError exception. Use the int() function if you want to get an integer, and float() if you need a floating-point value. Just like here:

si = '13'

sf = '1.3'

itg = int(si)

flt = float(sf)

print(itg + flt)

This is what you'll see in the console: 14.3

In the next section, we're going to show you some simple programs that process strings.

## 2.4.1.5 SECTION SUMMARY

**Key takeaways**

1. Strings can be compared to strings using general comparison operators, but comparing them to numbers gives no reasonable result, because **no string can be equal** to any number. For example:

* string == number is always False;
* string != number is always True;
* string >= number always **raises an exception.**

2. Sorting lists of strings can be done by:

* a function named sorted(), creating a new, sorted list;
* a method named sort(), which sorts the list in situ

3. A number can be converted to a string using the str() function.

4. A string can be converted to a number (although not every string) using either the int() or float() function. The conversion fails if a string doesn't contain a valid number image (an exception is raised then).

**Exercise 1: Which of the following lines describe a true condition?**

'smith' > 'Smith'

'Smiths' < 'Smith'

'Smith' > '1000'

'11' < '8'

Check: 1, 3 and 4

**Exercise 2: What is the expected output of the following code?**

s1 = 'Where are the snows of yesteryear?'

s2 = s1.split()

s3 = sorted(s2)

print(s3[1])

Check: are

**Exercise 3: What is the expected result of the following code?**

s1 = '12.8'

i = int(s1)

s2 = str(i)

f = float(s2)

print(s1 == s2)

Check: The code raises a ValueError exception

## 2.4.1.6 LAB: A LED Display

**LAB**

Estimated time: 30 minutes

Level of difficulty: Medium

**Objectives**

* improving the student's skills in operating with strings;
* using strings to represent non-text data.

**Scenario**

You've surely seen a seven-segment display. It's a device (sometimes electronic, sometimes mechanical) designed to present one decimal digit using a subset of seven segments. If you still don't know what it is, refer to the following Wikipedia article.

Your task is to write a program which is able to simulate the work of a seven-display device, although you're going to use single LEDs instead of segments.

Each digit is constructed from 13 LEDs (some lit, some dark, of course) - that's how we imagine it:

# ### ### # # ### ### ### ### ### ###

# # # # # # # # # # # # # #

# ### ### ### ### ### # ### ### # #

# # # # # # # # # # # # #

# ### ### # ### ### # ### ### ###

Note: the number 8 shows all the LED lights on. Your code has to display any non-negative integer number entered by the user. Tip: using a list containing patterns of all ten digits may be very helpful.

Test data

Sample input:123

Sample output:

# ### ###

# # #

# ### ###

# # #

# ### ###

Sample input: 9081726354

Sample output:

### ### ### # ### ### ### ### ### # #

# # # # # # # # # # # # # #

### # # ### # # ### ### ### ### ###

# # # # # # # # # # # # #

### ### ### # # ### ### ### ### #

# **2.5. STRINGS AND THE FOUR SIMPLE PROGRAMS**

## 2.5.1.1 Four simple programs:The Caesar Cipher: encrypting a message

We're going to show you four simple programs in order to present some aspects of string processing in Python. They are purposefully simple, but the lab problems will be significantly more complicated.

The first problem we want to show you is called the Caesar cipher - more details here: https://en.wikipedia.org/wiki/Caesar\_cipher.

This cipher was (probably) invented and used by Gaius Julius Caesar and his troops during the Gallic Wars. The idea is rather simple - every letter of the message is replaced by its nearest consequent (A becomes B, B becomes C, and so on). The only exception is Z, which becomes A. The program in the editor is a very simple (but working) implementation of the algorithm.

# Caesar cipher.

text = input("Enter your message: ")

cipher = ''

for char in text:

if not char.isalpha():

continue

char = char.upper()

code = ord(char) + 1

if code > ord('Z'):

code = ord('A')

cipher += chr(code)

print(cipher)

We've written it using the following assumptions:

* it accepts Latin letters only (note: the Romans used neither whitespaces nor digits)
* all letters of the message are in upper case (note: the Romans knew only capitals)

Let's trace the code:

* line 02: ask the user to enter the open (unencrypted), one-line message;
* line 03: prepare a string for an encrypted message (empty for now)
* line 04: start the iteration through the message;
* line 05: if the current character is not alphabetic...
* line 06: ...ignore it;
* line 07: convert the letter to upper-case (it's preferable to do it blindly, rather than check whether it's needed or not)
* line 08: get the code of the letter and increment it by one;
* line 09: if the resulting code has "left" the Latin alphabet (if it's greater than the Z code)...
* line 10: ...change it to the A code;
* line 11: append the received character to the end of the encrypted message;
* line 13: print the cipher.

The code, fed with this message: AVE CAESAR

outputs: BWFDBFTBS

## 2.5.1.2 Four simple programs: The Caesar Cipher: decrypting a message

The reverse transformation should now be clear to you - let's just present you with the code as-is, without any explanations. Look at the code in the editor. Check carefully if it works. Use the cryptogram from the previous program.

# Caesar cipher - decrypting a message.

cipher = input('Enter your cryptogram: ')

text = ''

for char in cipher:

if not char.isalpha():

continue

char = char.upper()

code = ord(char) - 1

if code < ord('A'):

code = ord('Z')

text += chr(code)

print(text)

## 2.5.1.3 Four simple programs: The Numbers Processor

The third program shows a simple method allowing you to input a line filled with numbers, and to process them easily. Note: the routine input() function, combined together with the int() or float() functions, is unsuitable for this purpose.

The processing will be extremely easy - we want the numbers to be summed. Look at the code in the editor. Let's analyze it.

# Numbers Processor.

line = input("Enter a line of numbers - separate them with spaces: ")

strings = line.split()

total = 0

try:

for substr in strings:

total += float(substr)

print("The total is:", total)

except:

print(substr, "is not a number.")

Using list comprehension may make the code slimmer. You can do that if you want.

Let's present our version:

* line 03: ask the user to enter a line filled with any number of numbers (the numbers can be floats)
* line 04: split the line receiving a list of substrings;
* line 05: initiate the total sum to zero;
* line 06: as the string-float conversion may raise an exception, it's best to continue with the protection of the try-except block;
* line 07: iterate through the list...
* line 08: ...and try to convert all its elements into float numbers; if it works, increase the sum;
* line 09: everything is good so far, so print the sum;
* line 10: the program ends here in the case of an error;
* line 11: print a diagnostic message showing the user the reason for the failure.

The code has one important weakness - it displays a bogus result when the user enters an empty line. Can you fix it?

## 2.5.1.4 Four simple programs: The IBAN Validator

The fourth program implements (in a slightly simplified form) an algorithm used by European banks to specify account numbers. The standard named **IBAN** (International Bank Account Number) provides a simple and fairly reliable method of validating the account numbers against simple typos that can occur during rewriting of the number e.g., from paper documents, like invoices or bills, into computers. Find details at: https://en.wikipedia.org/wiki/International\_Bank\_Account\_Number.

An IBAN-compliant account number consists of:

* a two-letter country code taken from the ISO 3166-1 standard (e.g., FR for France, GB for Great Britain, DE for Germany, and so on)
* two check digits used to perform the validity checks - fast and simple, but not fully reliable, tests, showing whether a number is invalid (distorted by a typo) or seems to be good;
* the actual account number (up to 30 alphanumeric characters - the length of that part depends on the country)

The standard says that validation requires the following steps (according to Wikipedia):

* (step 1) Check that the total IBAN length is correct as per the country (this program won't do that, but you can modify the code to meet this requirement if you wish; note: you have to teach the code all the lengths used in Europe)
* (step 2) Move the four initial characters to the end of the string (i.e., the country code and the check digits)
* (step 3) Replace each letter in the string with two digits, thereby expanding the string, where A = 10, B = 11 ... Z = 35;
* (step 4) Interpret the string as a decimal integer and compute the remainder of that number on division by 97; If the remainder is 1, the check digit test is passed and the IBAN might be valid.

Look at the code in the editor.

# IBAN Validator.

iban = input("Enter IBAN, please: ")

iban = iban.replace(' ','')

if not iban.isalnum():

print("You have entered invalid characters.")

elif len(iban) < 15:

print("IBAN entered is too short.")

elif len(iban) > 31:

print("IBAN entered is too long.")

else:

iban = (iban[4:] + iban[0:4]).upper()

iban2 = ''

for ch in iban:

if ch.isdigit():

iban2 += ch

else:

iban2 += str(10 + ord(ch) - ord('A'))

iban = int(iban2)

if iban % 97 == 1:

print("IBAN entered is valid.")

else:

print("IBAN entered is invalid.")

Let's analyze it:

* line 03: ask the user to enter the IBAN (the number can contain spaces, as they significantly improve number readability...
* line 04: ...but remove them immediately)
* line 05: the entered IBAN must consist of digits and letters only - if it doesn't...
* line 06: ...output the message;
* line 07: the IBAN mustn't be shorter than 15 characters (this is the shortest variant, used in Norway)
* line 08: if it is shorter, the user is informed;
* line 09: moreover, the IBAN cannot be longer than 31 characters (this is the longest variant, used in Malta)
* line 10: if it is longer, make an announcement;
* line 11: start the actual processing;
* line 12: move the four initial characters to the number's end, and convert all letters to upper case (step 02 of the algorithm)
* line 13: this is the variable used to complete the number, created by replacing the letters with digits (according to the algorithm's step 03)
* line 14: iterate through the IBAN;
* line 15: if the character is a digit...
* line 16: just copy it;
* line 17: otherwise...
* line 18: ...convert it into two digits (note the way it's done here)
* line 19: the converted form of the IBAN is ready - make an integer out of it;
* line 20: is the remainder of the division of iban2 by 97 equal to 1?
* line 21: If yes, then success;
* line 22: Otherwise...
* line 23: ...the number is invalid.

Let's add some test data (all these numbers are valid - you can invalidate them by changing any character).

British: GB72 HBZU 7006 7212 1253 00

French: FR76 30003 03620 00020216907 50

German: DE02100100100152517108

If you are an EU resident, you can use you own account number for tests.

## 2.5.1.5 SECTION SUMMARY

Key takeaways

1. Strings are key tools in modern data processing, as most useful data are actually strings. For example, using a web search engine (which seems quite trivial these days) utilizes extremely complex and complicated string processing, involving unimaginable amounts of data.

2. Comparing strings in a strict way (as Python does) can be very unsatisfactory when it comes to advanced searches (e.g. during extensive database queries). Responding to this demand, a number of fuzzy string comparison algorithms has been created and implemented. These algorithms are able to find strings which aren't equal in the Python sense, but are **similar**.

One such concept is the **Hamming distance**, which is used to determine the similarity of two strings. If this problem interests you, you can find out more about it here: https://en.wikipedia.org/wiki/Hamming\_distance. Another solution of the same kind, but based on a different assumption, is the **Levenshtein distance** described here: https://en.wikipedia.org/wiki/Levenshtein\_distance.

3. Another way of comparing strings is finding their acoustic similarity, which means a process leading to determine if two strings sound similar (like "raise" and "race"). Such a similarity has to be established for every language (or even dialect) separately.

An algorithm used to perform such a comparison for the English language is called **Soundex** and was invented – you won't believe – in 1918. You can find out more about it here: https://en.wikipedia.org/wiki/Soundex.

4. Due to limited native float and integer data precision, it's sometimes reasonable to store and process huge numeric values as strings. This is the technique Python uses when you force it to operate on an integer number consisting of a very large number of digits.

## 2.5.1.6 LAB: Improving the Caesar cipher 12/31/21

**LAB**

Estimated time: 30-90 minutes

Level of difficulty: Hard

**Pre-requisites**

Module 1.11.1.1, Module 1.11.1.2

**Objectives**

* + improving the student's skills in operating with strings;
  + converting characters into ASCII code, and vice versa.

**Scenario**

You are already familiar with the Caesar cipher, and this is why we want you to improve the code we showed you recently. The original Caesar cipher shifts each character by one: a becomes b, z becomes a, and so on. Let's make it a bit harder, and allow the shifted value to come from the range 1..25 inclusive.

Moreover, let the code preserve the letters' case (lower-case letters will remain lower-case) and all non-alphabetical characters should remain untouched. Your task is to write a program which:

* asks the user for one line of text to encrypt;
* asks the user for a shift value (an integer number from the range 1..25 - note: you should force the user to enter a valid shift value (don't give up and don't let bad data fool you!)
* prints out the encoded text.

Test your code using the data we've provided.

**Test data**

Sample input:

abcxyzABCxyz 123

2

Sample output:

cdezabCDEzab 123

Sample input:

The die is cast

25

Sample output:

Sgd chd hr bzrs

<https://itcosmos.co/improving-the-caesar-cipher-with-python/>

checkText = False

while checkText != True:

text = input("Enter your message: ")

if text != '':

checkText = True

checkShift = False

shiftNumber = list(range(1,26))

while checkShift != True:

try:

shift = int(input("Enter your shift number(1~25): "))

if shift in shiftNumber:

checkShift = True

except:

print("Please Enter number between 1 and 25!!")

continue

cipher = ''

for char in text:

if not char.isalpha():

cipher += char

if char.islower():

code = ord(char) + shift

if code > ord('z'):

code = ord('a') + (code - ord('z') - 1)

cipher += chr(code)

if char.isupper():

code = ord(char) + shift

if code > ord('Z'):

code = ord('A') + (code -ord('Z') - 1)

cipher += chr(code)

print(cipher)

## 2.5.1.7 LAB: Palindromes (To be worked on)

## 2.5.1.8 LAB: Anagrams (To be worked on)

## 2.5.1.9 LAB: The Digit of Life(To be worked on)

## 2.5.1.10 LAB: Find a word!(To be worked on)

## 2.5.1.11 LAB: Find a word!(To be worked on)

# 2.6. ERRORS - THE PROGRAMMER'S DAILY BREAD

## **2.6.1.1 Errors - the programmer's daily bread**

**Errors, failures, and other plagues**

Anything that can go wrong, will go wrong.

This is Murphy's law, and it works everywhere and always. Your code's execution can go wrong, too. If it can, it will.

Look the code in the editor. There are at least two possible ways it can "go wrong". Can you see them?

import math

x = float(input("Enter x: "))

y = math.sqrt(x)

print("The square root of", x, "equals to", y)

* As a user is able to enter a completely arbitrary string of characters, **there is no guarantee that the string can be converted into a float value** - this is the first vulnerability of the code;
* the second is that the sqrt() **function fails if it gets a negative argument.**

You may get one of the following error messages. Something like this:

Enter x: Abracadabra

Traceback (most recent call last):

File "sqrt.py", line 3, in <module>

x = float(input("Enter x: "))

ValueError: could not convert string to float: 'Abracadabra'

Or something like this:

Enter x: -1

Traceback (most recent call last):

File "sqrt.py", line 4, in <module>

y = math.sqrt(x)

ValueError: math domain error

Can you protect yourself from such surprises? Of course you can. Moreover, you have to do it in order to be considered a good programmer.

## **2.6.1.2 Errors - the programmer's daily bread: Exceptions**

Each time your code tries to do something wrong, foolish, irresponsible, crazy, unenforceable, Python does two things:

* it **stops your program;**
* it creates a special kind of data, called an **exception**.

Both of these activities are called **raising an exception**. We can say that Python always raises an exception (or that an **exception has been raised**) when it has no idea what to do with your code.

What happens next?

* the raised exception expects somebody/something to notice it and take care of it;
* if nothing happens to take care of the raised exception, the program will be **forcibly terminated**, and you will see an error message;
* otherwise, if the exception is taken care of and **handled** properly, the suspended program can be resumed and its execution can continue.

Python provides effective tools that allow you to **observe exceptions, identify them and handle them efficiently**. This is possible due to the fact that all potential exceptions have their unambiguous names, so you can categorize them and react appropriately. You know some exception names already. Take a look at the following diagnostic message: *ValueError: math domain error*

The word highlighted above is just the exception name. Let's get familiar with some other exceptions.

## **2.6.1.3 Errors - the programmer's daily bread: Exceptions: continued**

Look at the code in the editor. Run the (obviously incorrect) program.

value = 1

value /= 0

You will see the following message in reply:

Traceback (most recent call last):

File "div.py", line 2, in

value /= 0

ZeroDivisionError: division by zero

This exception error is called **ZeroDivisionError**.

## **2.6.1.4 Errors - the programmer's daily bread: Exceptions: continued**

Look at the code in the editor. What will happen when you run it? Check.

my\_list = []

x = my\_list[0]

You will see the following message in reply:

Traceback (most recent call last):

File "lst.py", line 2, in

x = list[0]

IndexError: list index out of range

This is the **IndexError**.

## **2.6.1.5 Errors - the programmer's daily bread: Exceptions: continued**

How do you **handle** exceptions? The word try is key to the solution. What's more, it's a keyword, too. The recipe for success is as follows:

* first, you have to **try to do something;**
* next, you have to **check whether everything went well.**

But wouldn't it be better to check all circumstances first and then do something only if it's safe? Just like the example in the editor.

first\_number = int(input("Enter the first number: "))

second\_number = int(input("Enter the second number: "))

if second\_number != 0:

print(first\_number / second\_number)

else:

print("This operation cannot be done.")

print("THE END.")

Admittedly, this way may seem to be the most natural and understandable, but in reality, this method doesn't make programming any easier. All these checks can make your code **bloated and illegible.**

Python prefers a completely different approach.

## **2.6.1.6** Errors - the programmer's daily bread | try-except

**Exceptions: continued**

Look at the code in the editor. This is the favorite Python approach.

first\_number = int(input("Enter the first number: "))

second\_number = int(input("Enter the second number: "))

try:

print(first\_number / second\_number)

except:

print("This operation cannot be done.")

print("THE END.")

Note:

* the try keyword **begins a block of the code** which may or may not be performing correctly;
* next, Python tries to perform the risky action; if it fails, an exception is raised and Python starts to look for a solution;
* the except keyword starts a piece of code which will be **executed if anything inside the try block goes wrong** - if an exception is raised inside a previous try block, **it will fail here**, so the code located after the except keyword should provide an **adequate reaction** to the raised exception;
* returning to the previous nesting level ends the **try-except section.**

Run the code and test its behavior.

Let's summarize this:

try:

:

:

except:

:

:

* in the first step, Python tries to perform all instructions placed between the try: and except: statements;
* if nothing is wrong with the execution and all instructions are performed successfully, the execution jumps to the point after the last line of the except: block, and the block's execution is considered complete;
* if anything goes wrong inside the try: and except: block, the execution immediately jumps out of the block and into the first instruction located after the except: keyword; this means that some of the instructions from the block may be silently omitted.

## **2.6.1.7** Errors - the programmer's daily bread | try-except:

Exceptions: continued

Look at the code in the editor. It will help you understand this mechanism.

try:

print("1")

x = 1 / 0

print("2")

except:

print("Oh dear, something went wrong...")

print("3")

This is the output it produces:

1

Oh dear, something went wrong...

3

Note: the print("2") instruction was lost in the process.

## **2.6.1.8** Errors - the programmer's daily bread | try-except

**Exceptions: continued**

This approach has one important disadvantage - if there is a possibility that more than one exception may skip into an except: branch, you may have **trouble figuring out what actually happened.** Just like in our code in the editor:

try:

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

y = 1 / x

except:

print("Oh dear, something went wrong...")

print("THE END.")

The message: *Oh dear, something went wrong...* appearing in the console says nothing about the reason, while there are two possible causes of the exception:

* non-integer data entered by the user;
* an integer value equal to 0 assigned to the x variable.

Technically, there are two ways to solve the issue:

* build two consecutive try-except blocks, one for each possible exception reason (easy, but will cause unfavorable code growth)
* use a more advanced variant of the instruction.

It looks like this:

try:

:

except exc1:

:

except exc2:

:

except:

:

This is how it works:

* if the try branch raises the exc1 exception, it will be handled by the except exc1: block;
* similarly, if the try branch raises the exc2 exception, it will be handled by the except exc2: block;
* if the try branch raises any other exception, it will be handled by the unnamed except block.

Let's move on to the next part of the course and see it in action.

## **2.6.1.9** Errors - the programmer's daily bread | try-except

**Exceptions: continued**

Look at the code in the editor. Our solution is there.

try:

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

y = 1 / x

print(y)

except ZeroDivisionError:

print("You cannot divide by zero, sorry.")

except ValueError:

print("You must enter an integer value.")

except:

print("Oh dear, something went wrong...")

print("THE END.")

The code, when run, produces one of the following four variants of output:

if you enter a valid, non-zero integer value (e.g., 5) it says:

*0.2*

*THE END.*

if you enter 0, it says:

*You cannot divide by zero, sorry.*

*THE END.*

if you enter any non-integer string, you see:

*You must enter an integer value.*

*THE END.*

(locally on your machine) if you press Ctrl-C while the program is waiting for the user's input (which causes an exception named KeyboardInterrupt), the program says:

Oh dear, something went wrong...

THE END.

## **2.6.1.10** Errors - the programmer's daily bread | try-except:

**Exceptions: continued**

Don't forget that:

* the *except* branches are searched in the same order as they appear in the code;
* you must not use more than one except branch with a certain exception name;
* the number of different *except* branches is arbitrary - the only condition is that if you use *try*, you must put at least one *except* (named or not) after it;
* the *except* keyword must not be used without a preceding *try*;
* if any of the *except* branches is executed, no other branches will be visited;
* if none of the specified *except* branches matches the raised exception, the exception remains unhandled (we'll discuss it soon)
* if an unnamed *except* branch exists (one without an exception name), it has to be specified as the last.

try:

:

except exc1:

:

except exc2:

:

except:

:

Let's continue the experiments now.

Look at the code in the editor. We've modified the previous program - we've removed the ZeroDivisionError branch.

try:

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

y = 1 / x

print(y)

except ValueError:

print("You must enter an integer value.")

except:

print("Oh dear, something went wrong...")

print("THE END.")

What happens now if the user enters 0 as an input? As there are no dedicated branches for division by zero, the raised exception falls into the general (unnamed) branch; this means that in this case, the program will say:

Oh dear, something went wrong...

THE END.

Try it yourself. Run the program.

## **2.6.1.11** Errors - the programmer's daily bread | try-except:

**Exceptions: continued**

Let's spoil the code once again. Look at the program in the editor. This time, we've removed the unnamed branch.

try:

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

y = 1 / x

print(y)

except ValueError:

print("You must enter an integer value.")

print("THE END.")

The user enters 0 once again and:

* the exception raised won't be handled by ValueError - it has nothing to do with it;
* as there's no other branch, you should to see this message:

Traceback (most recent call last):

File "exc.py", line 3, in

y = 1 / x

ZeroDivisionError: division by zero

You've learned a lot about exception handling in Python. In the next section, we will focus on Python built-in exceptions and their hierarchies.

## **2.6.1.12** Errors - the programmer's daily bread | try-except:

**Key takeaways**

1. An exception is an event in a program execution's life caused by an abnormal situation. The exception should he handled to avoid program termination. The part of your code that is suspected of being the source of the exception should be put inside the *try* branch. When the exception happens, the execution of the code is not terminated, but instead jumps into the *except* branch. This is the place where the handling of the exception should take place. The general scheme for such a construction looks as follows:

:

# The code that always runs smoothly.

:

try:

:

# Risky code.

:

except:

:

# Crisis management takes place here.

:

:

# Back to normal.

:

2. If you need to handle more than one exception coming from the same *try* branch ,you can add more than one *except* branch, but you have to label them with different exception names, like this:

:

# The code that always runs smoothly.

:

try:

:

# Risky code.

:

except Except\_1:

# Crisis management takes place here.

except Except\_2:

# We save the world here.

:

# Back to normal.

:

At most, one of the *except* branches is executed – none of the branches is performed when the raised exception doesn't match to the specified exceptions.

3. You cannot add more than one anonymous (unnamed) except branch after the named ones.

:

# The code that always runs smoothly.

:

try:

:

# Risky code.

:

except Except\_1:

# Crisis management takes place here.

except Except\_2:

# We save the world here.

except:

# All other issues fall here.

:

# Back to normal.

:

**Exercise 1:** What is the expected output of the following code?

try:

print("Let's try to do this")

print("#"[2])

print("We succeeded!")

except:

print("We failed")

print("We're done")

Check

Let's try to do this

We failed

We're done

**Exercise 2: What is the expected output of the following code?**

try:

print("alpha"[1/0])

except ZeroDivisionError:

print("zero")

except IndexingError:

print("index")

except:

print("some")

Check

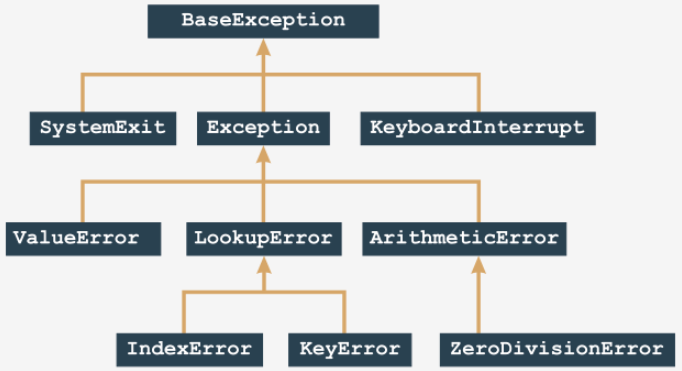
zero

# 2.7. THE HIERARCHY OF EXCEPTIONS

## **2.7.1.1 The anatomy of exceptions: Exceptions**

Python 3 defines **63 built-in exceptions**, and all of them form a **tree-shaped hierarchy**, although the tree is a bit weird as its root is located on top.

Some of the built-in exceptions are more general (they include other exceptions) while others are completely concrete (they represent themselves only). We can say that t**he closer to the root an exception is located, the more general (abstract) it is**. In turn, the exceptions located at the branches' ends (we can call them **leaves**) are concrete. Take a look at the figure:

It shows a small section of the complete exception tree. Let's begin examining the tree from the ZeroDivisionError leaf.

Note:

* *ZeroDivisionError* is a special case of a general exception class: ArithmeticError;
* *ArithmeticError* is a special case of a general exception class: Exception;
* *Exception* is a special case of a more general class named BaseException;

We can describe it in the following way (note the direction of the arrows - they always point to the more general entity):

BaseException

↑

Exception

↑

ArithmeticError

↑

ZeroDivisionError

We're going to show you how this generalization works. Let's start with some really simple code.

## **2.7.1.2 The anatomy of exceptions:** Exceptions: continued

Look at the code in the editor. It is a simple example to start with. Run it.

try:

y = 1 / 0

except ZeroDivisionError:

print("Oooppsss...")

print("THE END.")

output

Oooppsss...

THE END.

Now look at the code below:

try:

y = 1 / 0

except ArithmeticError:

print("Oooppsss...")

print("THE END.")

Something has changed in it - we've replaced ZeroDivisionError with ArithmeticError. You already know that ArithmeticError is a general class including (among others) the ZeroDivisionError exception. Thus, the code's output remains unchanged. Test it.

This also means that replacing the exception's name with either Exception or BaseException won't change the program's behavior.

Let's summarize:

* each exception raised **falls into the first matching branch;**
* the matching branch doesn't have to specify the same exception exactly - it's enough that the exception is **more general** (more abstract) than the raised one.

## **2.7.1.3 The anatomy of exceptions:** Exceptions: continued

Look at the code in the editor. What will happen here?

try:

y = 1 / 0

except ZeroDivisionError:

print("Zero Division!")

except ArithmeticError:

print("Arithmetic problem!")

print("THE END.")

The first matching branch is the one containing ZeroDivisionError. It means that the console will show:

Zero division!

THE END.

Will it change anything if we swap the two except branches around? like here below:

try:

y = 1 / 0

except ArithmeticError:

print("Arithmetic problem!")

except ZeroDivisionError:

print("Zero Division!")

print("THE END.")

The change is radical - the code's output is now:

Arithmetic problem!

THE END.

Why, if the exception raised is the same as previously?

The exception is the same, but the more general exception is now listed first - it will catch all zero divisions too. It also means that there's no chance that any exception hits the ZeroDivisionError branch. This branch is now completely unreachable. Remember:

* the order of the branches matters!
* don't put more general exceptions before more concrete ones;
* this will make the latter one unreachable and useless;
* moreover, it will make your code messy and inconsistent;
* Python won't generate any error messages regarding this issue.

## **2.7.1.4 The anatomy of exceptions:** Exceptions: continued 1/1/2022

If you want to **handle two or more exceptions** in the same way, you can use the following syntax:

try:

:

except (exc1, exc2):

:

You simply have to put all the engaged exception names into a comma-separated list with parentheses. If an **exception is raised inside a function**, it can be handled:

* inside the function;
* outside the function;

Let's start with the first variant - look at the code in the editor.

def bad\_fun(n):

try:

return 1 / n

except ArithmeticError:

print("Arithmetic Problem!")

return None

bad\_fun(0)

print("THE END.")

The *ZeroDivisionError* exception (being a concrete case of the *ArithmeticError* exception class) is raised inside the bad\_fun() function, and it doesn't leave the function - the function itself takes care of it. The program outputs:

Arithmetic problem!

THE END.

It's also possible to let the exception propagate outside the function. Let's test it now.

Look at the code below:

def bad\_fun(n):

return 1 / n

try:

bad\_fun(0)

except ArithmeticError:

print("What happened? An exception was raised!")

print("THE END.")

The problem has to be solved by the invoker (or by the invoker's invoker, and so on).

The program outputs:

What happened? An exception was raised!

THE END.

Note: **the exception raised can cross function and module boundaries**, and travel through the invocation chain looking for a matching *except* clause able to handle it.

If there is no such clause, the exception remains unhandled, and Python solves the problem in its standard way - **by terminating your code and emitting a diagnostic message.** Now we're going to suspend this discussion, as we want to introduce you to a brand new Python instruction.

## **2.7.1.5 The anatomy of exceptions | raise:** Exceptions: continued

The *raise* instruction raises the specified exception named *exc* as if it was raised in a normal (natural) way:

raise exc

Note: raise is a keyword.

The instruction enables you to:

* **simulate raising actual exceptions** (e.g., to test your handling strategy)
* partially **handle an exception** and make another part of the code responsible for completing the handling (separation of concerns).

Look at the code in the editor. This is how you can use it in practice.

def bad\_fun(n):

raise ZeroDivisionError

try:

bad\_fun(0)

except ArithmeticError:

print("What happened? An error?")

print("THE END.")

The program's output remains unchanged.

What happened? An error?

THE END.

In this way, you can **test your exception handling routine** without forcing the code to do stupid things.

## **2.7.1.6 The anatomy of exceptions | raise:** Exceptions: continued

The raise instruction may also be utilized in the following way (note the absence of the exception's name):

raise

There is one serious restriction: this kind of raise instruction may be used **inside the except branch only**; using it in any other context causes an error.

The instruction will immediately re-raise the same exception as currently handled. Thanks to this, you can distribute the exception handling among different parts of the code. Look at the code in the editor. Run it - we'll see it in action.

def bad\_fun(n):

try:

return n / 0

except:

print("I did it again!")

raise

try:

bad\_fun(0)

except ArithmeticError:

print("I see!")

print("THE END.")

The ZeroDivisionError is raised twice:

* first, inside the try part of the code (this is caused by actual zero division)
* second, inside the except part by the raise instruction.

In effect, the code outputs:

I did it again!

I see!

THE END.

## **2.7.1.7 The anatomy of exceptions | raise:** Exceptions: continued

Now is a good moment to show you another Python instruction, named assert. This is a keyword.

assert expression

How does it work?

* It evaluates the expression;
* if the expression evaluates to True, or a non-zero numerical value, or a non-empty string, or any other value different than None, it won't do anything else;
* otherwise, it automatically and immediately raises an exception named AssertionError (in this case, we say that the assertion has failed)

How it can be used?

* you may want to put it into your code where you want to be a**bsolutely safe from evidently wrong data**, and where you aren't absolutely sure that the data has been carefully examined before (e.g., inside a function used by someone else)
* raising an *AssertionError* exception secures your code from producing invalid results, and clearly shows the nature of the failure;
* **assertions don't supersede exceptions or validate the data** - they are their supplements.

If exceptions and data validation are like careful driving, assertion can play the role of an airbag.Let's see the assert instruction in action. Look at the code in the editor. Run it.

import math

x = float(input("Enter a number: "))

assert x >= 0.0

x = math.sqrt(x)

print(x)

The program runs flawlessly if you enter a valid numerical value greater than or equal to zero; otherwise, it stops and emits the following message:

Traceback (most recent call last):

File ".main.py", line 4, in

assert x >= 0.0

AssertionError

## 2.7.1.8 SECTION SUMMARY

**Key takeaways**

1. You cannot add more than one anonymous (unnamed) except branch after the named ones.

# The code that always runs smoothly.

:

try:

:

# Risky code.

:

except Except\_1:

# Crisis management takes place here.

except Except\_2:

# We save the world here.

except:

# All other issues fall here.

:

# Back to normal.

2. All the predefined Python exceptions form a hierarchy, i.e. some of them are more general (the one named BaseException is the most general one) while others are more or less concrete (e.g. IndexError is more concrete than LookupError).

You shouldn't put more concrete exceptions before the more general ones inside the same except branche sequence. For example, you can do this:

try:

# Risky code.

except IndexError:

# Taking care of mistreated lists

except LookupError:

# Dealing with other erroneous lookups

but don't do that (unless you're absolutely sure that you want some part of your code to be useless)

try:

# Risky code.

except LookupError:

# Dealing with erroneous lookups

except IndexError:

# You'll never get here

3. The Python statement raise ExceptionName can raise an exception on demand. The same statement, but lacking ExceptionName, can be used inside the try branch **only**, and raises the same exception which is currently being handled.

4. The Python statement assert expression evaluates the expression and raises the AssertError exception when the expression is equal to zero, an empty string, or None. You can use it to protect some critical parts of your code from devastating data.

**Exercise 1: What is the expected output of the following code?**

try:

print(1/0)

except ZeroDivisionError:

print("zero")

except ArithmeticError:

print("arith")

except:

print("some")

Check: zero

**Exercise 2: What is the expected output of the following code?**

try:

print(1/0)

except ArithmeticError:

print("arith")

except ZeroDivisionError:

print("zero")

except:

print("some")

Check: arith

**Exercise 3: What is the expected output of the following code?**

def foo(x):

assert x

return 1/x

try:

print(foo(0))

except ZeroDivisionError:

print("zero")

except:

print("some")

Check: some

# 2.8. USEFUL EXCEPTIONS

## **2.8.1.1 Useful exceptions:** Built-in exceptions: **ArithmeticError, AssertionError**

We're going to show you a short list of the most useful exceptions. While it may sound strange to call "useful" a thing or a phenomenon which is a visible sign of failure or setback, as you know, to err is human and if anything can go wrong, it will go wrong. Exceptions are as routine and normal as any other aspect of a programmer's life. For each exception, we'll show you:

* its name;
* its location in the exception tree;
* a short description;
* a concise snippet of code showing the circumstances in which the exception may be raised.

There are lots of other exceptions to explore - we simply don't have the space to go through them all here.

**ArithmeticError**

**Location**: BaseException ← Exception ← ArithmeticError

**Description**: an abstract exception including all exceptions caused by arithmetic operations like zero division or an argument's invalid domain

**AssertionError**

**Location**: BaseException ← Exception ← AssertionError

**Description**: a concrete exception raised by the assert instruction when its argument evaluates to False, None, 0, or an empty string

**Code**:

from math import tan, radians

angle = int(input('Enter integral angle in degrees: '))

# We must be sure that angle != 90 + k \* 180

assert angle % 180 != 90

print(tan(radians(angle)))

**BaseException**

**Location**: BaseException

**Description**: the most general (abstract) of all Python exceptions - all other exceptions are included in this one; it can be said that the following two except branches are equivalent: *except:* and *except BaseException:*.

**IndexError**

**Location**: BaseException ← Exception ← LookupError ← IndexError

**Description**: a concrete exception raised when you try to access a non-existent sequence's element (e.g., a list's element). Code:

# The code shows an extravagant way

# of leaving the loop.

the\_list = [1, 2, 3, 4, 5]

ix = 0

do\_it = True

while do\_it:

try:

print(the\_list[ix])

ix += 1

except IndexError:

do\_it = False

print('Done')

Output:

1

2

3

4

5

Done

## **2.8.1.2 Useful exceptions: KeyboardInterrupt, LookupError, MemoryError,**

**KeyboardInterrupt**

**Location**: BaseException ← KeyboardInterrupt

**Description**: a concrete exception raised when the user uses a keyboard shortcut designed to terminate a program's execution (Ctrl-C in most OSs); if handling this exception doesn't lead to program termination, the program continues its execution.

Note: this exception is not derived from the Exception class. Run the program：

# This code cannot be terminated

# by pressing Ctrl-C.

from time import sleep

seconds = 0

while True:

try:

print(seconds)

seconds += 1

sleep(1)

except KeyboardInterrupt:

print("Don't do that!")

Output:

0

1

2

:

:

：

**LookupError**

**Location**: BaseException ← Exception ← LookupError

**Description**: an abstract exception including all exceptions caused by errors resulting from invalid references to different collections (lists, tuples, etc.)

**MemoryError**

**Location**: BaseException ← Exception ← MemoryError

**Description**: a concrete exception raised when an operation cannot be completed due to a lack of free memory.

Code:

# This code causes the MemoryError exception.

# Warning: executing this code may affect your OS.

# Don't run it in production environments!

string = 'x'

try:

while True:

string = string + string

print(len(string))

except MemoryError:

print('This is not funny!')

**OverflowError**

**Location**: BaseException ← Exception ← ArithmeticError ← OverflowError

**Description**: a concrete exception raised when an operation produces a number too big to be successfully stored.

# The code prints subsequent

# values of exp(k), k = 1, 2, 4, 8, 16, ...

from math import exp

ex = 1

try:

while True:

print(exp(ex))

ex \*= 2

except OverflowError:

print('The number is too big.')

## **2.8.1.3 Useful exceptions: ImportError and KeyError**

**ImportError**

**Location**: BaseException ← Exception ← StandardError ← ImportError

**Description**: a concrete exception raised when an import operation fails

# One of these imports will fail - which one?

try:

import math

import time

import abracadabra

except:

print('One of your imports has failed.')

**KeyError**

**Location**: BaseException ← Exception ← LookupError ← KeyError

**Description**: a concrete exception raised when you try to access a collection's non-existent element (e.g., a dictionary's element)

Code:

# How to abuse the dictionary

# and how to deal with it?

dictionary = { 'a': 'b', 'b': 'c', 'c': 'd' }

ch = 'a'

try:

while True:

ch = dictionary[ch]

print(ch)

except KeyError:

print('No such key:', ch)

We are done with exceptions for now, but they'll return when we discuss object-oriented programming in Python. You can use them to protect your code from bad accidents, but you also have to learn how to dive into them, exploring the information they carry.

Exceptions are in fact objects - however, we can tell you nothing about this aspect until we present you with classes, objects, and the like. For the time being, if you'd like to learn more about exceptions on your own, you look into Standard Python Library at <https://docs.python.org/3.6/library/exceptions.html.>

## **2.8.1.4 Reading ints safely**

**LAB**

Estimated time: 15-25 minutes

Level of difficulty: Medium

**Objectives**

* improving the student's skills in defining functions;
* using exceptions in order to provide a safe input environment.

**Scenario**

Your task is to write a **function able to input integer values and to check if they are within a specified range.** The function should:

* accept 3 arguments: a prompt, a low acceptable limit, and a high acceptable limit;
* if the user enters a string that is not an integer value, the function should emit the message Error: wrong input, and ask the user to input the value again;
* if the user enters a number which falls outside the specified range, the function should emit the message Error: the value is not within permitted range (min..max) and ask the user to input the value again;
* if the input value is valid, return it as a result.

**Test data**

Test your code carefully.

This is how the function should react to the user's input:

Enter a number from -10 to 10: 100

Error: the value is not within permitted range (-10..10)

Enter a number from -10 to 10: asd

Error: wrong input

Enter number from -10 to 10: 1

The number is: 1

## **2.8.1.5 SECTION SUMMARY**

Key takeaways

1. Some abstract built-in Python exceptions are:

* ArithmeticError,
* BaseException,
* LookupError.

2. Some concrete built-in Python exceptions are:

* AssertionError,
* ImportError,
* IndexError,
* KeyboardInterrupt,
* KeyError,
* MemoryError,
* OverflowError.

**Exercise 1:** Which of the exceptions will you use to protect your code from being interrupted through the use of the keyboard?

Check:KeyboardInterrupt

**Exercise 2**: What is the name of the most general of all Python exceptions?

Check: BaseException

**Exercise 3**: Which of the exceptions will be raised through the following unsuccessful evaluation?

huge\_value = 1E250 \*\* 2

Check: OverflowError

## **2.8.1.6 Module Completion**

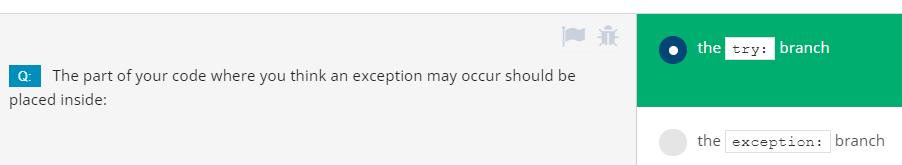
Congratulations! You have completed PE2: Module 2.

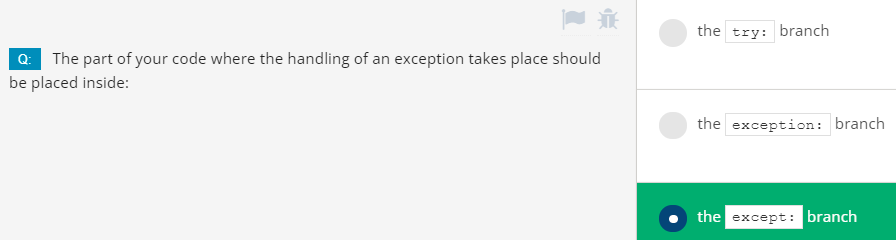
Well done! You've reached the end of Module 2 and completed a major milestone in your Python programming education. Here's a short summary of the objectives you've covered and got familiar with in Module 2:

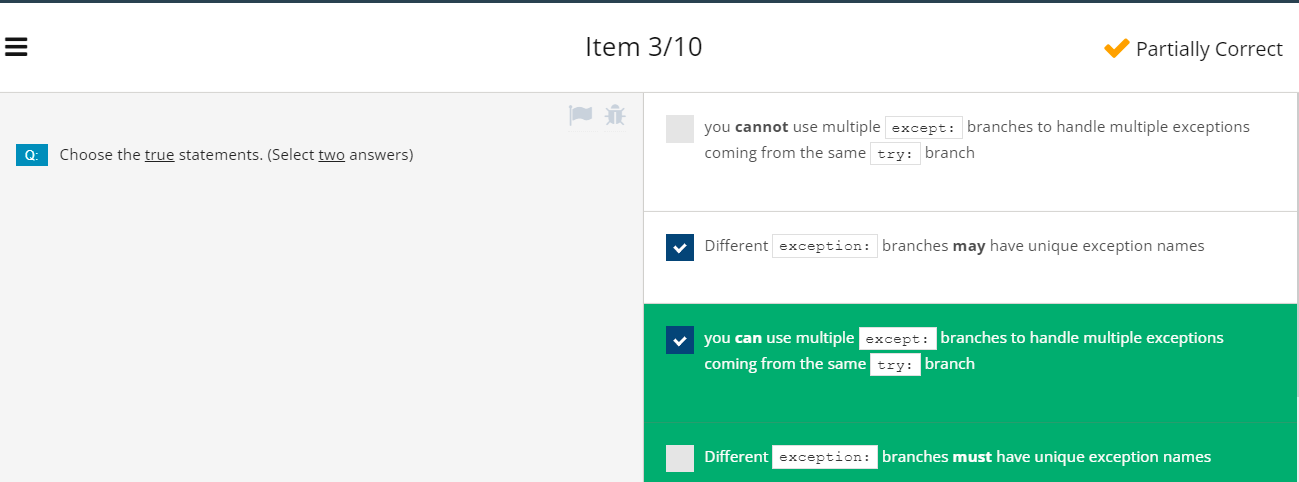
* characters, strings, and coding standards;
* the nature of strings in Python; strings vs. lists - similarities and differences;
* list and string methods;
* handling errors in Python;
* controlling the flow of errors using try and except;
* the hierarchy of exceptions; review of the most useful exceptions.

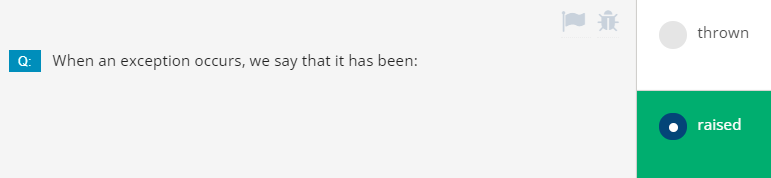
You are now ready to take the module quiz and attempt the final challenge: Module 2 Test, which will help you gauge what you've learned so far.

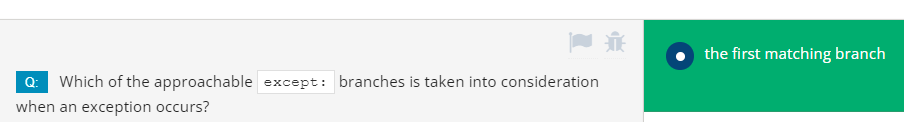
# 2.9 PE2 Module 2 Quiz 1/2/2022

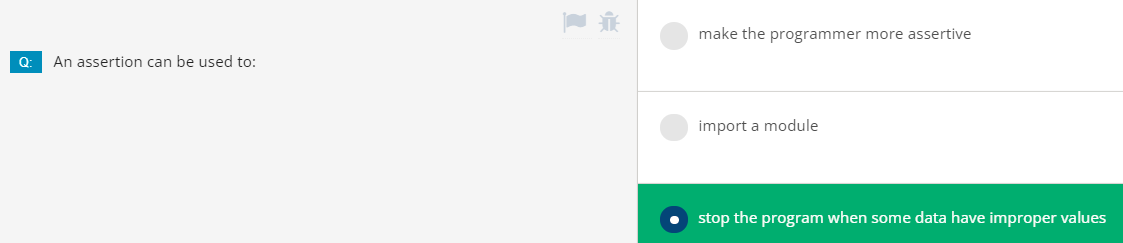


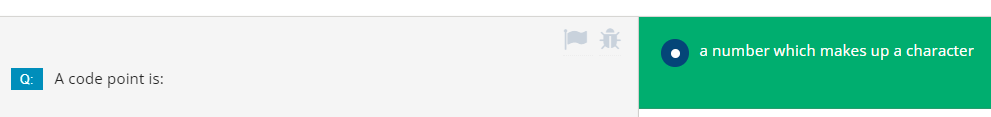


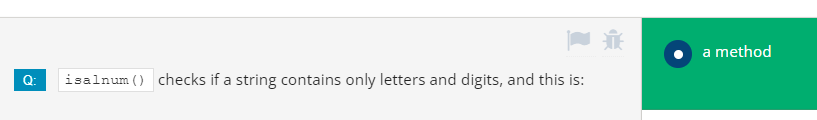


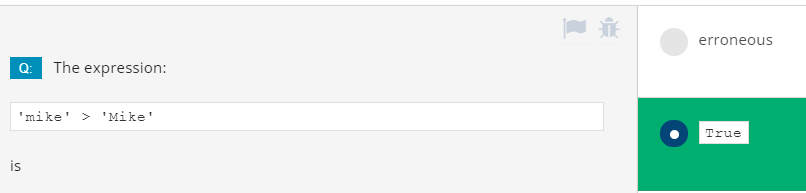


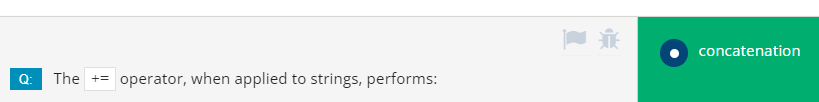




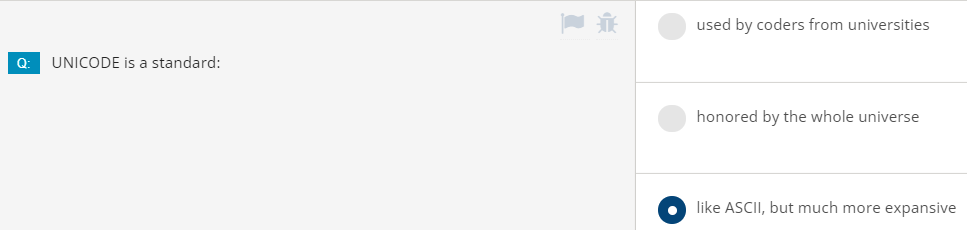


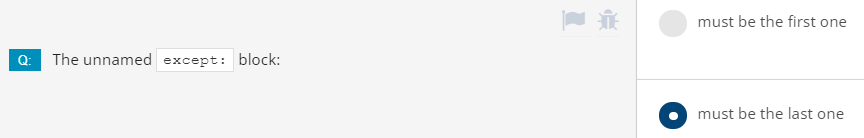


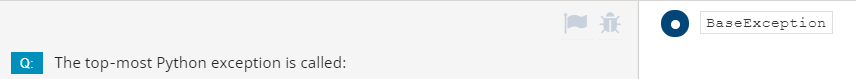


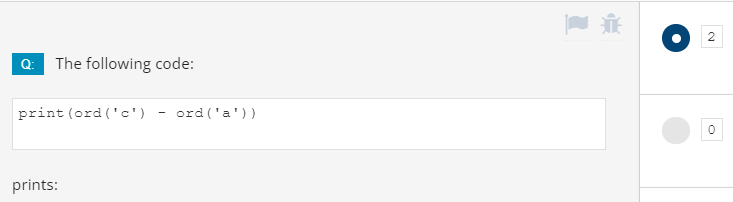


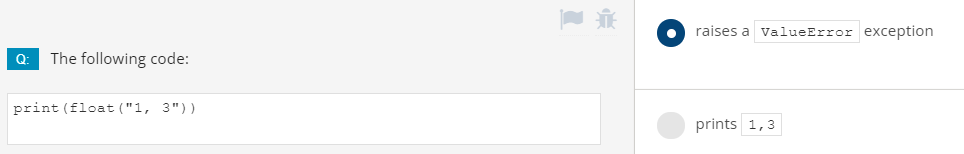
# Module 2 Test (with all correct answers)



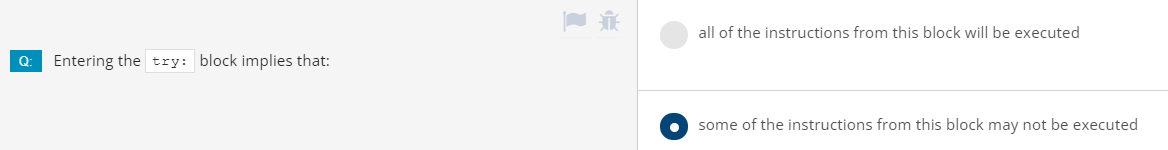


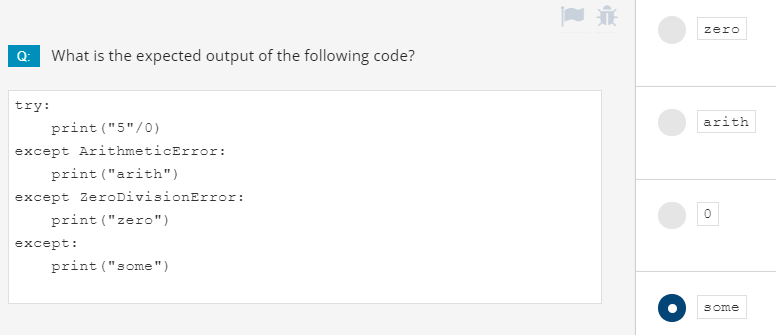


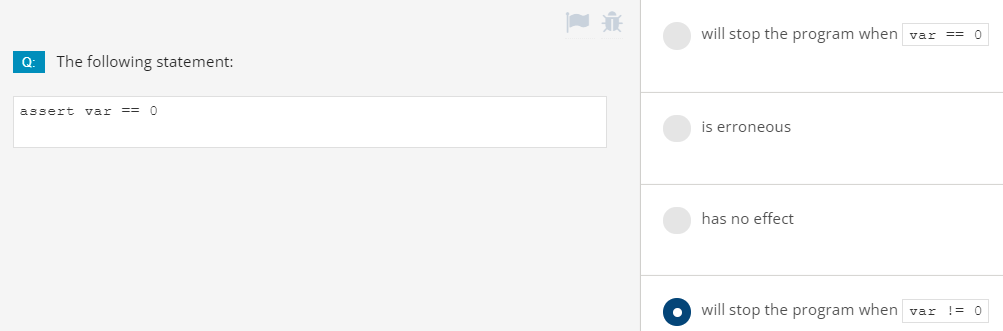


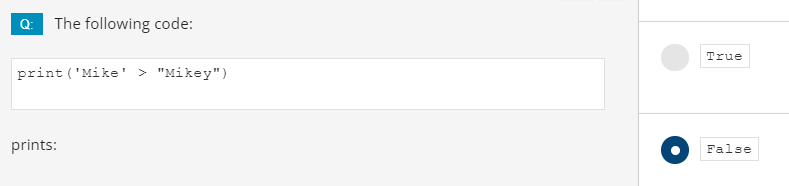


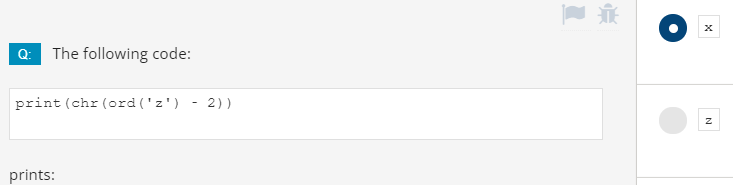


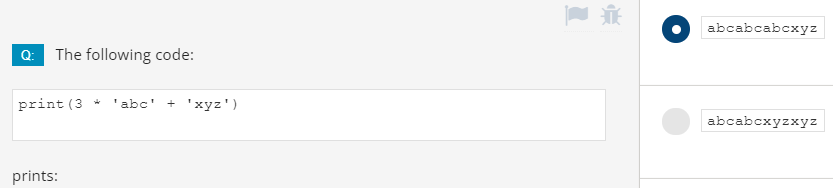




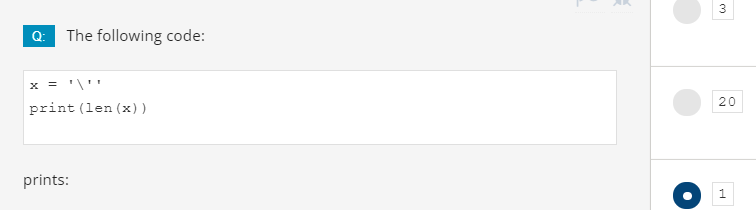


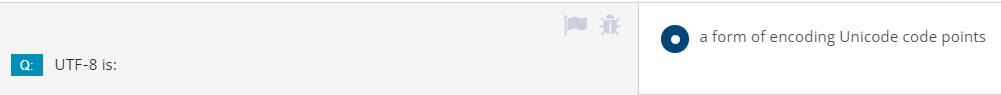












# **MODULE 3: Object-Oriented Programming**

# **3.1. THE FOUNDATIONS OF OOP (CLASSES, OBJECTS, ATTRIBUTES)**

## **3.1.1.0 Python Essentials 2 - Module 3**

**Python Essentials 2: Module 3 Object-Oriented Programming**

In this module, you will learn about:

* Basic concepts of object-oriented programming (OOP)
* Differences between the procedural & object approaches (motivations & profits)
* Classes, objects, properties, and methods;
* Designing reusable classes and creating objects;
* Inheritance and polymorphism;
* Exceptions as objects.

## **3.1.1.1 The foundations of OOP**

**The basic concepts of the object-oriented approach**

Let's take a step outside of computer programming and computers in general, and discuss object programming issues.

Nearly all of the programs and techniques you have used till now fall under the procedural style of programming. Admittedly, you have made use of some built-in objects, but when referring to them, we just mentioned the absolute minimum.

The procedural style of programming was the dominant approach to software development for decades of IT, and it is still in use today. Moreover, it isn't going to disappear in the future, as it works very well for specific types of projects (generally, not very complex ones and not large ones, but there are lots of exceptions to that rule).

The object approach is quite young (much younger than the procedural approach) and is particularly useful when applied to big and complex projects carried out by large teams consisting of many developers.

This kind of understanding of a project's structure makes many important tasks easier, e.g., dividing the project into small, independent parts, and independent development of different project elements.

**Python is a universal tool for both object and procedural programming**. It may be successfully utilized in both spheres.

Furthermore, you can create lots of useful applications, even if you know nothing about classes and objects, but you have to keep in mind that some of the problems (e.g., graphical user interface handling) may require a strict object approach. Fortunately, object programming is relatively simple.

## **3.1.1.2 The foundations of OOP**

**Procedural vs. the object-oriented approach**

In the **procedural approach**, it's possible to distinguish two different and completely separate worlds: **the world of data**, and **the world of code**. The world of data is populated with variables of different kinds, while the world of code is inhabited by code grouped into modules and functions.

Functions are able to use data, but not vice versa. Furthermore, functions are able to abuse data, i.e., to use the value in an unauthorized manner (e.g., when the sine function gets a bank account balance as a parameter).

We said in the past that data cannot use functions. But is this entirely true? Are there some special kinds of data that can use functions?

Yes, there are - the ones named methods. These are functions which are invoked from within the data, not beside them. If you can see this distinction, you've taken the first step into object programming.

**The object approach** suggests a completely different way of thinking. The data and the code are enclosed together in the same world, divided into classes.

Every **class is like a recipe which can be used when you want to create a useful object** (this is where the name of the approach comes from). You may produce as many objects as you need to solve your problem.

Every object has a set of traits (they are called properties or attributes - we'll use both words synonymously) and is able to perform a set of activities (which are called methods).

The recipes may be modified if they are inadequate for specific purposes and, in effect, new classes may be created. These new classes inherit properties and methods from the originals, and usually add some new ones, creating new, more specific tools.

**Objects are incarnations(化身)** of ideas expressed in classes, like a cheesecake on your plate is an incarnation of the idea expressed in a recipe printed in an old cookbook. The objects interact with each other, exchanging data or activating their methods. A properly constructed class (and thus, its objects) are able to protect the sensible data and hide it from unauthorized modifications.

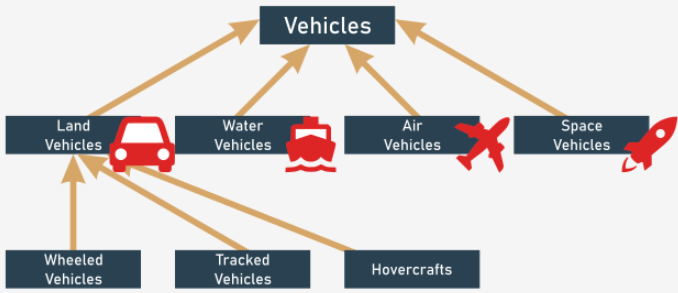
There is no clear border between data and code: they live as one in objects.

All these concepts are not as abstract as you may at first suspect. On the contrary, they all are taken from real-life experiences, and therefore are extremely useful in computer programming: they don't create artificial life - **they reflect real facts, relationships, and circumstances.**

## **3.1.1.3 The foundations of OOP:** Class hierarchies

The word class has many meanings, but not all of them are compatible with the ideas we want to discuss here. The class that we are concerned with is like a category, as a result of precisely defined similarities.We'll try to point out a few classes which are good examples of this concept.

The concept of class hierarchies: vehicles

Let's look for a moment at vehicles. All existing vehicles (and those that don't exist yet) are **related by a single, important feature**: the ability to move. You may argue that a dog moves, too; is a dog a vehicle? No, it isn't. We have to improve the definition, i.e., enrich it with other criteria, distinguishing vehicles from other beings, and creating a stronger connection. Let's take the following circumstances into consideration: vehicles are artificially created entities used for transportation, moved by forces of nature, and directed (driven) by humans.

Based on this definition, a dog is not a vehicle. The *vehicles* class is very broad. Too broad. We have to define some more **specialized classes**, then. The specialized classes are the **subclasses**. The vehicles class will be a **superclass** for them all.

Note: **the hierarchy grows from top to bottom, like tree roots, not branches**. The most general, and the widest, class is always at the top (the superclass) while its descendants are located below (the subclasses).

By now, you can probably point out some potential subclasses for the Vehicles superclass. There are many possible classifications. We've chosen subclasses based on the environment, and say that there are (at least) four subclasses:

* land vehicles;
* water vehicles;
* air vehicles;
* space vehicles.

In this example, we'll discuss the first subclass only - land vehicles. If you wish, you can continue with the remaining classes.

Land vehicles may be further divided, depending on the method with which they impact the ground. So, we can enumerate:

* wheeled vehicles;
* tracked vehicles;
* hovercrafts.

The hierarchy we've created is illustrated by the figure.

Note the direction of the arrows - they always point to the superclass. The top-level class is an exception - it doesn't have its own superclass.

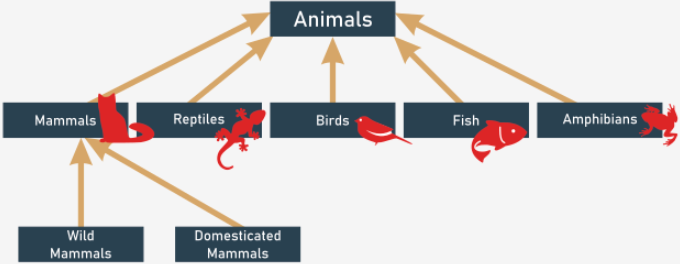
## **3.1.1.4 The foundations of OOP:** Class hierarchies continued

Another example is the hierarchy of the taxonomic kingdom of animals. We can say that all animals (our top-level class) can be divided into five subclasses:

* mammals;
* reptiles;
* birds;
* fish;
* amphibians.

We'll take the first one further. We have identified the following subclasses:

* wild mammals;
* domesticated mammals.

Try to extend the hierarchy any way you want, and find the right place for humans.

## **3.1.1.5 The foundations of OOP:** What is an object?

A class (among other definitions) is a **set of objects**. An object is a **being belonging to a class.** An object is **an incarnation of the requirements, traits, and qualities assigned to a specific class.** This may sound simple, but note the following important circumstances. Classes form a hierarchy.

This may mean that an object belonging to a specific class belongs to all the superclasses at the same time. It may also mean that any object belonging to a superclass may not belong to any of its subclasses.

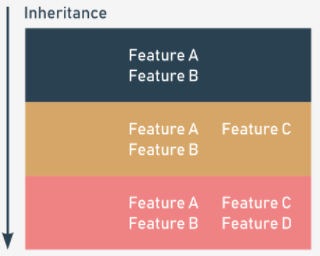
For example: any personal car is an object belonging to the wheeled vehicles class. It also means that the same car belongs to all superclasses of its home class; therefore, it is a member of the vehicles class, too.

Your dog (or your cat) is an object included in the domesticated mammals class, which explicitly means that it is included in the animals class as well. Each **subclass is more specialized** (or more specific) than its superclass. Conversely, each **superclass is more general** (more abstract) than any of its subclasses.

Note that we've presumed that a class may only have one superclass - this is not always true, but we'll discuss this issue more a bit later.

**Inheritance**

Let's define one of the fundamental concepts of object programming, named **inheritance**. Any object bound to a specific level of a class hierarchy **inherits all the traits (as well as the requirements and qualities) defined inside any of the superclasses.** The object's home class may define new traits (as well as requirements and qualities) which will be inherited by any of its subclasses.



The inheritance concept

You shouldn't have any problems matching this rule to specific examples, whether it applies to animals, or to vehicles.

## **3.1.1.6 The foundations of OOP:** What does an object have?

The object programming convention assumes that **every existing object may be equipped with three groups of attributes:**

* an object has a **name** that uniquely identifies it within its home namespace (although there may be some anonymous objects, too)
* an object has **a set of individual properties** which make it original, unique, or outstanding (although it's possible that some objects may have no properties)
* an object has **a set of abilities to perform specific activities**, able to change the object itself, or some of the other objects.

There is a hint (although this doesn't always work) which can help you identify any of the three spheres above. Whenever you describe an object and you use:

* a noun – you probably define the object's name;
* an adjective – you probably define the object's property;
* a verb – you probably define the object's activity.

Two sample phrases should serve as a good example:

* A pink Cadillac went quickly.

Object name = Cadillac

Home class = Wheeled vehicles

Property = Color (pink)

Activity = Go (quickly)

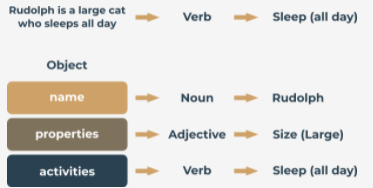
* Rudolph is a large cat sleeps all day.

Object name = Rudolph

Home class = Cat

Property = Size (large)

Activity = Sleep (all day)



## **3.1.1.7 The foundations of OOP:** Your first class

Object programming is **the art of defining and expanding classes**. A class is a model of a very specific part of reality, reflecting properties and activities found in the real world. The classes defined at the beginning are too general and imprecise to cover the largest possible number of real cases. There's no obstacle to defining new, more precise subclasses. They'll inherit everything from their superclass, so the work that went into its creation isn't wasted.

The new class may add new properties and new activities, and therefore may be more useful in specific applications. Obviously, it may be used as a superclass for any number of newly created subclasses. The process doesn't need to have an end. You can create as many classes as you need.

The class you define has nothing to do with the object: **the existence of a class does not mean that any of the compatible objects will automatically be created.** The class itself isn't able to create an object - you have to create it yourself, and Python allows you to do this. It's time to define the simplest class and to create an object. Take a look at the example below:

class TheSimplestClass:

pass

We've defined a class there. The class is rather poor: it has neither properties nor activities. It's **empty**, actually, but that doesn't matter for now. The simpler the class, the better for our purposes.

**The definition begins with the keyword** *class*. The keyword is followed by an **identifier which will name the class** (note: don't confuse it with the object's name - these are two different things).

Next, you add a **colon** (:), as classes, like functions, form their own nested block. The content inside the block define all the class's properties and activities. The *pass* keyword fills the class with nothing. It doesn't contain any methods or properties.

**Your first object**

The newly defined class becomes a tool that is able to create new objects. The tool has to be used explicitly, on demand. Imagine that you want to create one (exactly one) object of the TheSimplestClass class.

To do this, you need to assign a variable to store the newly created object of that class, and create an object at the same time. You do it in the following way:

my\_first\_object = TheSimplestClass()

Note:

* the class name tries to pretend that it's a function. We'll discuss it soon;
* the newly created object is equipped with everything the class brings; as this class is completely empty, the object is empty, too.

The act of creating an object of the selected class is also called an **instantiation** (as the object becomes an **instance of the class**).

Let's leave classes alone for a short moment, as we're now going to tell you a few words about *stacks*. We know the concept of classes and objects may not be fully clear yet. Don't worry, we'll explain everything very soon.

## **3.1.1.8 SECTION SUMMARY**

**Key takeaways**

1. A **class** is an idea (more or less abstract) which can be used to create a number of incarnations – such an incarnation is called an **object**.

2. When a class is derived from another class, their relation is named **inheritance**. The class which derives from the other class is named a **subclass**. The second side of this relation is named **superclass**. A way to present such a relation is an **inheritance diagram**, where:

* superclasses are always presented above their subclasses;
* relations between classes are shown as arrows directed **from the subclass toward its superclass**.

3. Objects are equipped with:

* a **name** which identifies them and allows us to distinguish between them;
* a set of **properties** (the set can be empty)
* a set of **methods** (can be empty, too)

4. To define a Python class, you need to use the *class* keyword. For example:

class This\_Is\_A\_Class:

pass

5. To create an object of the previously defined class, you need to use the class as if it were a function. For example:

this\_is\_an\_object = This\_Is\_A\_Class()

**Exercise 1:**If we assume that pythons, vipers, and cobras are subclasses of the same superclass, how would you call it?

Check:Snake, reptile, vertebrate, animal – all these answers are acceptable.

**Exercise 2:** Try to name a few python class subclasses.

Check: Indian python, African rock python, ball python – the list is long.

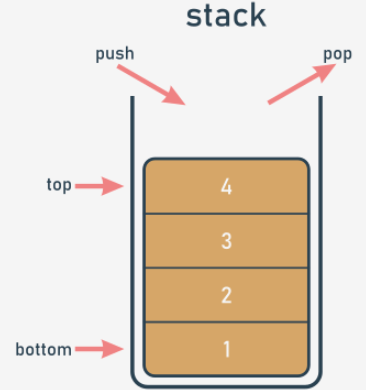
**Exercise 3:** Can you name one of your classes just "class"?

Check: No, you can't – class is a keyword!

# 3.2. STACK: THE PROCEDURAL VS OOP APPROACH

## **3.2.1.1 A short journey from procedural to object approach：What is a stack?**

**A stack is a structure developed to store data in a very specific way**. Imagine a stack of coins. You aren't able to put a coin anywhere else but on the top of the stack. Similarly, you can't get a coin off the stack from any place other than the top of the stack. If you want to get the coin that lies on the bottom, you have to remove all the coins from the higher levels.

The alternative name for a stack (but only in IT terminology) is **LIFO**. It's an abbreviation for a very clear description of the stack's behavior: **Last In - First Out**. The coin that came last onto the stack will leave first.

**A stack is an object** with two elementary operations, conventionally named **push** (when a new element is put on the top) and **pop** (when an existing element is taken away from the top).

Stacks are used very often in many classical algorithms, and it's hard to imagine the implementation of many widely used tools without the use of stacks.

Let's implement a stack in Python. This will be a very simple stack, and we'll show you how to do it in two independent approaches: procedural and objective. Let's start with the first one.

## **3.2.1.2 A short journey: The stack - the procedural approach**

First, you have to decide how to store the values which will arrive onto the stack. We suggest using the simplest of methods, and **employing a list** for this job. Let's assume that the size of the stack is not limited in any way. Let's also assume that the last element of the list stores the top element. The stack itself is already created:

stack = []

We're ready to **define a function that puts a value onto the stack**. Here are the presuppositions(假设) for it:

* the name for the function is *push*;
* the function gets one parameter (this is the value to be put onto the stack)
* the function returns nothing;
* the function appends the parameter's value to the end of the stack;

This is how we've done it - take a look:

def push(val):

stack.append(val)

Now it's time for a **function to take a value off the stack**. This is how you can do it:

* the name of the function is pop;
* the function doesn't get any parameters;
* the function returns the value taken from the stack
* the function reads the value from the top of the stack and removes it.

The function is here:

def pop():

val = stack[-1]

del stack[-1]

return val

Note: the function doesn't check if there is any element in the stack.

Let's assemble all the pieces together to set the stack in motion. The **complete program** pushes three numbers onto the stack, pulls them off, and prints their values on the screen. You can see it in the editor window.

stack = []

def push(val):

stack.append(val)

def pop():

val = stack[-1]

del stack[-1]

return val

push(3)

push(2)

push(1)

print(pop())

print(pop())

print(pop())

outputs:

1

2

3

## **3.2.1.3 The stack - the procedural approach vs. the object-oriented approach 1/3/2022**

The procedural stack is ready. Of course, there are some weaknesses, and the implementation could be improved in many ways (harnessing exceptions to work is a good idea), but in general the stack is fully implemented, and you can use it if you need to. But the more often you use it, the more disadvantages you'll encounter. Here are some of them:

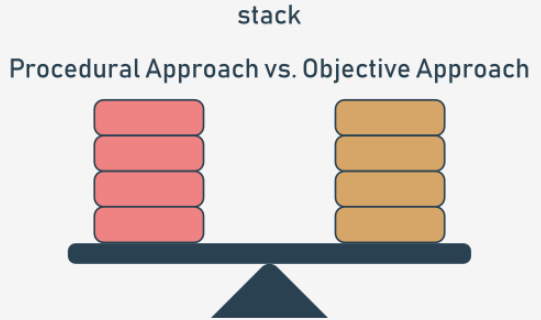
* the essential variable (the stack list) is highly **vulnerable**; anyone can modify it in an uncontrollable way, destroying the stack, in effect; this doesn't mean that it's been done maliciously - on the contrary, it may happen as a result of carelessness, e.g., when somebody confuses variable names; imagine that you have accidentally written something like this: stack[0] = 0

The functioning of the stack will be completely disorganized;

* it may also happen that one day you need more than one stack; you'll have to create another list for the stack's storage, and probably other *push* and *pop* functions too;
* it may also happen that you need not only *push* and *pop* functions, but also some other conveniences; you could certainly implement them, but try to imagine what would happen if you had dozens of separately implemented stacks.

The objective approach delivers solutions for each of the above problems. Let's name them first:

* the ability to hide (protect) selected values against unauthorized access is called **encapsulation**; **the encapsulated values can be neither accessed nor modified if you want to use them exclusively;**
* when you have a class implementing all the needed stack behaviors, you can produce as many stacks as you want; you needn't copy or replicate any part of the code;
* the ability to enrich the stack with new functions comes from inheritance; you can create a new class (a subclass) which inherits all the existing traits from the superclass, and adds some new ones.

Let's now write a brand new stack implementation from scratch. This time, we'll use the objective approach, guiding you step by step into the world of object programming.

## **3.2.1.4 The stack - the object approach 1/4/2022**

Of course, the main idea remains the same. We'll use a list as the stack's storage. We only have to know how to put the list into the class. Let's start from the absolute beginning - this is how the objective stack begins: class Stack:

Now, we expect two things from it:

* we want the class to have **one property as the stack's storage - we have to "install" a list inside each object of the class** (note: each object has to have its own list - the list mustn't be shared among different stacks)
* then, we want **the list to be hidden** from the class users' sight.

How is this done?

In contrast to other programming languages, Python has no means of allowing you to declare such a property just like that. Instead, you need to add a specific statement or instruction. The properties have to be added to the class manually. How do you guarantee that such an activity takes place every time the new stack is created?

There is a simple way to do it - you have to **equip the class with a specific function** - its specificity is dual:

* it has to be named in a strict way;
* it is invoked implicitly, when the new object is created.

Such a function is called a **constructor**, as its general purpose is to **construct a new object**. The constructor should know everything about the object's structure, and must perform all the needed initializations. Let's add a very simple constructor to the new class. Take a look at the snippet:

class Stack:

def \_\_init\_\_(self):

print("Hi!")

stack\_object = Stack()

And now:

* the constructor's name is always \_\_init\_\_;
* it has to have **at least one parameter** (discuss this later); the parameter is used to represent the newly created object - you can use the parameter to manipulate the object, and to enrich it with the needed properties; you'll make use of this soon;
* note: the obligatory parameter is usually named *self* - **it's only a convention, but you should follow it** to simplify the process of reading/ understanding your code.

The code is in the editor. Run it now. Here is its output: Hi!

Note - there is no trace of invoking the constructor inside the code. It has been invoked implicitly and automatically. Let's make use of that now.

## **3.2.1.5** The stack - the object approach: continued

Any change you make inside the constructor that modifies the state of the *self* parameter will be reflected in the newly created object. This means you can add any property to the object and the property will remain there until the object finishes its life or the property is explicitly removed. Now let's **add just one property to the new object** - a list for a stack. We'll name it *stack\_list*. Just like here:

class Stack:

def \_\_init\_\_(self):

self.stack\_list = []

stack\_object = Stack()

print(len(stack\_object.stack\_list))

Note:

* we've used the **dotted notation**, just like when invoking methods; this is the general convention for accessing an object's properties - you need to name the object, put a *dot* (.) after it, and specify the desired property's name; don't use parentheses! You don't want to invoke a method - you want to **access a property**;
* if you set a property's value for the very first time (like in the constructor), you are creating it; then, the object has got the property and is ready to use its value;
* we've done something more in the code - we've tried to access the *stack\_list* property from outside the class immediately after the object has been created; we want to check the current length of the stack - have we succeeded?

Yes, we have - the code produces the following output: 0

This is not we want from the stack. We prefer **stack\_list to be hidden from the outside world.** Is that possible? Yes, and it's simple, but not very intuitive.

## **3.2.1.6** The stack - the object approach: continued

Take a look - we've added two underscores before the stack\_list name - nothing more:

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

stack\_object = Stack()

print(len(stack\_object.\_\_stack\_list))

The change invalidates the program. Why? When any class component has a **name starting with two underscores (\_\_), it becomes private** - this means that it can be accessed only from within the class. You cannot see it from the outside world. This is how Python implements the **encapsulation** concept. Run the program to test our assumptions - an AttributeError exception should be raised.

Traceback (most recent call last):

File "main.py", line 7, in <module>

print(len(stack\_object.\_\_stack\_list))

AttributeError: 'Stack' object has no attribute '\_\_stack\_list'

## **3.2.1.7** The object approach: a stack from scratch

Now it's time for the two functions (methods) implementing the *push* and *pop* operations. Python assumes that a function of this kind (a class activity) should be **immersed inside the class body** - just like a constructor.

We want to invoke these functions to *push* and *pop* values. This means that they should both be accessible to every class's user (in contrast to the previously constructed list, which is hidden from the ordinary class's users).

Such a component is called **public**, so you **can't begin its name with two (or more) underscores.** There is one more requirement - **the name must have no more than one trailing underscore**. As no trailing underscores at all fully meets the requirement, you can assume that the name is acceptable.

The functions themselves are simple. Take a look:

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

def push(self, val):

self.\_\_stack\_list.append(val)

def pop(self):

val = self.\_\_stack\_list[-1]

del self.\_\_stack\_list[-1]

return val

stack\_object = Stack()

stack\_object.push(3)

stack\_object.push(2)

stack\_object.push(1)

print(stack\_object.pop())

print(stack\_object.pop())

print(stack\_object.pop())

However, there's something really strange in the code. The functions look familiar, but they have more parameters than their procedural counterparts. Here, both functions have a parameter named *self* at the first position of the parameters list. Is it needed? Yes, it is.

All methods have to have this parameter. It plays the same role as the first constructor parameter. **It allows the method to access entities (properties and activities/methods) carried out by the actual object.** You cannot omit it. Every time Python invokes a method, it implicitly sends the current object as the first argument.

This means that a **method is obligated to have at least one parameter, which is used by Python itself** - you don't have any influence on it.

If your method needs no parameters at all, this one must be specified anyway. If it's designed to process just one parameter, you have to specify two, and the first one's role is still the same. There is one more thing that requires explanation - the way in which methods are invoked from within the *\_\_stack\_list* variable.

Fortunately, it's much simpler than it looks:

* the first stage delivers the object as a whole → *self*;
* next, you need to get to the *\_\_stack\_list* list → *self.\_\_stack\_list*;
* with *\_\_stack\_list* ready to be used, you can perform the third and last step → *self.\_\_stack\_list.append(val)*.

The class declaration is complete, and all its components have been listed. The class is ready for use.

## **3.2.1.8** The object approach: a stack from scratch

Having such a class opens up some new possibilities. For example, you can now have more than one stack behaving in the same way. Each stack will have its own copy of private data, but will utilize the same set of methods. This is exactly what we want for this example. Analyze the code:

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

def push(self, val):

self.\_\_stack\_list.append(val)

def pop(self):

val = self.\_\_stack\_list[-1]

del self.\_\_stack\_list[-1]

return val

stack\_object\_1 = Stack()

stack\_object\_2 = Stack()

stack\_object\_1.push(3)

stack\_object\_2.push(stack\_object\_1.pop())

print(stack\_object\_2.pop())

There are **two stacks created from the same base class. They work independently.** You can make more of them if you want to. Run the code in the editor and see what happens. Carry out your own experiments.

## **3.2.1.9** The object approach: a stack from scratch (continued)

Analyze the snippet below - we've created three objects of the class *Stack*. Next, we've juggled them up. Try to predict the value outputted to the screen.

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

def push(self, val):

self.\_\_stack\_list.append(val)

def pop(self):

val = self.\_\_stack\_list[-1]

del self.\_\_stack\_list[-1]

return val

little\_stack = Stack()

another\_stack = Stack()

funny\_stack = Stack()

little\_stack.push(1)

another\_stack.push(little\_stack.pop() + 1)

funny\_stack.push(another\_stack.pop() - 2)

print(funny\_stack.pop())

So, what's the result? Run the program. Output:0. Check if you were right.

## **3.2.1.10** The object approach: a stack from scratch (continued)

Let's go further. We **add a new class for handling stacks.** The new class should be able to **evaluate the sum of all the elements currently stored on the stack.**

We don't want to modify the previously defined stack. It's already good enough in its applications, and we don't want it changed in any way. We want a new stack with new capabilities. In other words, we want to construct a subclass of the already existing Stack class. The first step is easy: **just define a new subclass pointing to the class which will be used as the superclass.** This is what it looks like:

class AddingStack(Stack):

pass

The class doesn't define any new component yet, but that doesn't mean that it's empty. **It gets all the components defined by its superclass** - the name of the superclass is written before the colon directly after the new class name.

This is what we want from the new stack:

* we want the *push* method not only to push the value onto the stack but also to add the value to the *sum* variable;
* we want the *pop* function not only to pop the value off the stack but also to subtract the value from the *sum* variable.

Firstly, let's add a new variable to the class. It'll be a **private variable**, like the stack list. We don't want anybody to manipulate the sum value.

As you already know, adding a new property to the class is done by the constructor. You already know how to do that, but there is something really intriguing inside the constructor. Take a look:

class AddingStack(Stack):

def \_\_init\_\_(self):

Stack.\_\_init\_\_(self)

self.\_\_sum = 0

The second line of the constructor's body creates a property named *\_\_sum* - it will store the total of all the stack's values. But the line before it looks different. What does it do? Is it really necessary? Yes, it is.

Contrary to many other languages, Python forces you to **explicitly invoke a superclass's constructor.** Omitting this point will have harmful effects - the object will be deprived of the *\_\_stack\_list* list. Such a stack will not function properly.

This is the only time you can invoke any of the available constructors explicitly - it can be done inside the subclass's constructor. Note the syntax:

* you specify the superclass's name (this is the class whose constructor you want to run). you put a dot (.)after it;
* you specify the name of the constructor;
* you have to point to the object (the class's instance) which has to be initialized by the constructor - this is why you have to specify the argument and use the self variable here; note: **invoking any method (including constructors) from outside the class never requires you to put the self argument at the argument's list** - invoking a method from within the class demands explicit usage of the self argument, and it has to be put first on the list.

Note: it's generally a recommended practice to invoke the superclass's constructor before any other initializations you want to perform inside the subclass. This is the rule we have followed in the snippet.

## **3.2.1.11** The object approach: a stack from scratch (continued) 1/6/22

Secondly, let's add two methods. But let us ask you: is it really adding? We have these methods in the superclass already. Can we do something like that?

Yes, we can. It means that we're going to **change the functionality of the methods, not their names.** We can say more precisely that the interface (the way in which the objects are handled) of the class remains the same when changing the implementation at the same time. Let's start with the implementation of the *push* function. This is what we expect from it:

* to add the value to the *\_\_sum* variable;
* to push the value onto the stack.

Note: the second activity has already been implemented inside the superclass - so we can use that. Furthermore, we have to use it, as there's no other way to access the *\_\_stackList* variable.

This is how the *push* method looks in the subclass:

def push(self, val):

self.\_\_sum += val

Stack.push(self, val)

Note the way we've invoked the previous implementation of the *push* method (the one available in the superclass):

* we have to specify the superclass's name; this is necessary in order to clearly indicate the class containing the method, to avoid confusing it with any other function of the same name;
* we have to specify the target object and to pass it as the first argument (it's not implicitly added to the invocation in this context.)

We say that the *push* method has been overridden - the same name as in the superclass now represents a different functionality.

## **3.2.1.12** The object approach: a stack from scratch (continued) 1/7/22

This is the new *pop* function:

def pop(self):

val = Stack.pop(self)

self.\_\_sum -= val

return val

So far, we've defined the *\_\_sum* variable, but we haven't provided a method to get its value. It seems to be hidden. How can we reveal it and do it in a way that still protects it from modifications?

We have to define a new method. We'll name it *get\_sum*. Its only task will be to **return the \_\_sum value.** Here it is:

def get\_sum(self):

return self.\_\_sum

So, let's look at the program in the editor. The complete code of the class is there. We can check its functioning now, and we do it with the help of a very few additional lines of code. As you can see, we add five subsequent values onto the stack, print their sum, and take them all off the stack.

Okay, this has been a very brief introduction to Python's object programming. Soon we're going to tell you about it all in more detail.

## **3.2.1.13 SECTION SUMMARY**

**Key takeaways**

1. A **stack** is an object designed to store data using the **LIFO** model. The stack usually accomplishes at least two operations, named **push**() and **pop**().

2. Implementing the stack in a procedural model raises several problems which can be solved by the techniques offered by **OOP** (Object Oriented Programming):

3. A class **method** is actually a function declared inside the class and able to access all the class's components.

4. The part of the Python class responsible for creating new objects is called the **constructor**, and it's implemented as a method of the name *\_\_init\_\_*.

5. Each class method declaration must contain at least one parameter (always the first one) usually referred to as *self*, and is used by the objects to identify themselves.

6. If we want to hide any of a class's components from the outside world, we should start its name with \_\_. Such components are called **private**.

**Exercise 1:** Assuming that there is a class named Snakes, write the very first line of the Python class declaration, expressing the fact that the new class is actually a subclass of Snake.

Check：class Python(Snakes):

**Exercise 2:** Something is missing from the following declaration – what?

class Snakes

def \_\_init\_\_():

self.sound = 'Sssssss'

Check: The \_\_init\_\_() constructor lacks the obligatory parameter (we should name it self to stay compliant with the standards).

**Exercise 3**: Modify the code to guarantee that the venomous property is private.

class Snakes

def \_\_init\_\_(self):

self.venomous = True

Check: The code should look as follows:

class Snakes

def \_\_init\_\_(self):

self.\_\_venomous = True

## **3.2.1.14 Counting stack (To be worked on)**

## **3.2.1.15 Queue aka FIFO (To be worked on)**

## **3.2.1.16 Queue aka FIFO: part 2 (To be worked on)**

# 3.3. PROPERTIES (INSTANCE VAR, CLASS VAR, ATTRIBUTES)

## **3.3.1.1 OOP: Properties:** Instance variables 1/8/22

In general, a class can be equipped with two different kinds of data to form a class's properties. You already saw one of them when we were looking at stacks.

This kind of class property exists when and only when it is explicitly created and added to an object. As you already know, this can be done during the object's initialization, performed by the constructor. Moreover, it can be done in any moment of the object's life. Furthermore, any existing property can be removed at any time. Such an approach has some important consequences:

* different objects of the same class **may possess different sets of properties**;
* there must be a way to **safely check if a specific object owns the property** you want to utilize (unless you want to provoke an exception - it's worth considering)
* each object **carries its own set of properties** - they don't interfere with one another in any way.

Such variables (properties) are called **instance variables**.

The word *instance* suggests that they are closely connected to the objects (which are class instances), not to the classes themselves. Let's take a closer look at them. Here is an example:

class ExampleClass:

def \_\_init\_\_(self, val = 1):

self.first = val

def set\_second(self, val):

self.second = val

example\_object\_1 = ExampleClass()

example\_object\_2 = ExampleClass(2)

example\_object\_2.set\_second(3)

example\_object\_3 = ExampleClass(4)

example\_object\_3.third = 5

print(example\_object\_1.\_\_dict\_\_)

print(example\_object\_2.\_\_dict\_\_)

print(example\_object\_3.\_\_dict\_\_)

It needs one additional explanation before we go into any more detail. Take a look at the last three lines of the code.

Python objects, when created, are gifted with a **small set of predefined properties and methods**. Each object has got them, whether you want them or not. One of them is a variable named \_\_dict\_\_ (it's a dictionary). The variable contains the names and values of all the properties (variables) the object is currently carrying. Let's make use of it to safely present an object's contents. Let's dive into the code now:

* the class named *ExampleClass* has a constructor, which **unconditionally creates an instance variable** named *first*, and sets it with the value passed through the first argument (from the class user's perspective) or the second argument (from the constructor's perspective); note the default value of the parameter - any trick you can do with a regular function parameter can be applied to methods, too;
* the class also has a **method which creates another instance variable:** *second*;
* we've created three objects of the class *ExampleClass*, but all these instances differ:
* *example\_object\_1* only has the property named *first*;
* *example\_object\_2* has two properties: *first* and *second*;
* *example\_object\_3* has been enriched with a property named *third* just on the fly, outside the class's code - this is possible and fully permissible.

The program's output clearly shows that our assumptions are correct - here it is:

{'first': 1}

{'second': 3, 'first': 2}

{'third': 5, 'first': 4}

There is one additional conclusion that should be stated here: **modifying an instance variable of any object has no impact on all the remaining objects**. Instance variables are perfectly isolated from each other.

## 3.3.1.2 OOP: Properties: Instance variables: continued

Take a look at the modified example in the editor.

class ExampleClass:

def \_\_init\_\_(self, val = 1):

self.\_\_first = val

def set\_second(self, val = 2):

self.\_\_second = val

example\_object\_1 = ExampleClass()

example\_object\_2 = ExampleClass(2)

example\_object\_2.set\_second(3)

example\_object\_3 = ExampleClass(4)

example\_object\_3.\_\_third = 5

print(example\_object\_1.\_\_dict\_\_)

print(example\_object\_2.\_\_dict\_\_)

print(example\_object\_3.\_\_dict\_\_)

It's nearly the same as the previous one. The only difference is in the property names. We've **added two underscores** (\_\_) in front of them. As you know, such an addition makes the instance variable **private** - it becomes inaccessible from the outer world. The actual behavior of these names is a bit more complicated, so let's run the program. This is the output:

{'\_ExampleClass\_\_first': 1}

{'\_ExampleClass\_\_first': 2, '\_ExampleClass\_\_second': 3}

{'\_ExampleClass\_\_first': 4, '\_\_third': 5}

Can you see these strange names full of underscores? Where did they come from?

When Python sees that you want to add an instance variable to an object and you're going to do it inside any of the object's methods, it **mangles the operation** in the following way:

* it puts a class name before your name;
* it puts an additional underscore at the beginning.

This is why the *\_\_first* becomes *\_ExampleClass\_\_first*. **The name is now fully accessible from outside the class**. You can run a code like this:

print(example\_object\_1.\_ExampleClass\_\_first)

and you'll get a valid result with no errors or exceptions.

As you can see, making a property private is limited. **The mangling won't work if you add a private instance variable outside the class code.** In this case, it'll behave like any other ordinary property.

## 3.3.1.3 OOP: Properties: Class variables

A class variable is **a property which exists in just one copy and is stored outside any object**. Note: no instance variable exists if there is no object in the class; a class variable exists in one copy even if there are no objects in the class. Class variables are created differently to their instance siblings. The example will tell you more:

class ExampleClass:

counter = 0

def \_\_init\_\_(self, val = 1):

self.\_\_first = val

ExampleClass.counter += 1

example\_object\_1 = ExampleClass()

example\_object\_2 = ExampleClass(2)

example\_object\_3 = ExampleClass(4)

print(example\_object\_1.\_\_dict\_\_, example\_object\_1.counter)

print(example\_object\_2.\_\_dict\_\_, example\_object\_2.counter)

print(example\_object\_3.\_\_dict\_\_, example\_object\_3.counter)

Look:

* there is an assignment in the first list of the class definition - it sets the variable named counter to 0; initializing the variable inside the class but outside any of its methods makes the variable a class variable;
* accessing such a variable looks the same as accessing any instance attribute - you can see it in the constructor body; as you can see, the constructor increments the variable by one; in effect, the variable counts all the created objects.

Running the code will cause the following output:

{'\_ExampleClass\_\_first': 1} 3

{'\_ExampleClass\_\_first': 2} 3

{'\_ExampleClass\_\_first': 4} 3

Two important conclusions come from the example:

* class variables **aren't shown in an object's** \_\_dict\_\_ (this is natural as class variables aren't parts of an object) but you can always try to look into the variable of the same name, but at the class level - we'll show you this very soon;
* a class variable **always presents the same value** in all class instances (objects)

## 3.3.1.4 OOP: Properties: Class variables: continued

Mangling a class variable's name has the same effects as those you're already familiar with.Look at the example in the editor. Can you guess its output?

class ExampleClass:

\_\_counter = 0

def \_\_init\_\_(self, val = 1):

self.\_\_first = val

ExampleClass.\_\_counter += 1

example\_object\_1 = ExampleClass()

example\_object\_2 = ExampleClass(2)

example\_object\_3 = ExampleClass(4)

print(example\_object\_1.\_\_dict\_\_, example\_object\_1.\_ExampleClass\_\_counter)

print(example\_object\_2.\_\_dict\_\_, example\_object\_2.\_ExampleClass\_\_counter)

print(example\_object\_3.\_\_dict\_\_, example\_object\_3.\_ExampleClass\_\_counter)

Run the program and check if your predictions were correct. Everything works as expected, doesn't it?

{'\_ExampleClass\_\_first': 1} 3

{'\_ExampleClass\_\_first': 2} 3

{'\_ExampleClass\_\_first': 4} 3

## 3.3.1.5 OOP: Properties: Class variables: continued

We told you before that class variables exist even when no class instance (object) had been created. Now we're going to take the opportunity to show you **the differe**nce **between these two \_\_dict\_\_ variables**, the one from the class and the one from the object. Look at the code in the editor. The proof is there.

class ExampleClass:

varia = 1

def \_\_init\_\_(self, val):

ExampleClass.varia = val

print(ExampleClass.\_\_dict\_\_)

example\_object = ExampleClass(2)

print(ExampleClass.\_\_dict\_\_)

print(example\_object.\_\_dict\_\_)

Let's take a closer look at it:

1. We define one class named *ExampleClass*;
2. The class defines one class variable named *varia*;
3. The class constructor sets the variable with the parameter's value;
4. Naming the variable is the most important aspect of the example because:
   1. Changing the assignment to *self.varia = val* would create an instance variable of the same name as the class's one;
   2. Changing the assignment to *varia = val* would operate on a method's local variable; (we strongly encourage you to test both of the above cases - this will make it easier for you to remember the difference)
5. The first line of the off-class code prints the value of the ExampleClass.varia attribute; note - we use the value before the very first object of the class is instantiated.

Run the code in the editor and check its output.

{'\_\_module\_\_': '\_\_main\_\_', 'varia': 1, '\_\_init\_\_': <function ExampleClass.\_\_init\_\_ at 0x7ff1b35830e0>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'ExampleClass' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'ExampleClass' objects>, '\_\_doc\_\_': None}

{'\_\_module\_\_': '\_\_main\_\_', 'varia': 2, '\_\_init\_\_': <function ExampleClass.\_\_init\_\_ at 0x7ff1b35830e0>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'ExampleClass' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'ExampleClass' objects>, '\_\_doc\_\_': None}

{}

As you can see, the class' \_\_dict\_\_ contains much more data than its object's counterpart. Most of them are useless now - the one we want you to check carefully shows the current varia value.

Note that the object's \_\_dict\_\_ is empty - the object has no instance variables.

## 3.3.1.6 OOP: Properties: Checking an attribute's existence

Python's attitude to object instantiation raises one important issue - in contrast to other programming languages, **you may not expect that all objects of the same class have the same sets of properties.**

Just like in the example in the editor. Look at it carefully.

class ExampleClass:

def \_\_init\_\_(self, val):

if val % 2 != 0:

self.a = 1

else:

self.b = 1

example\_object = ExampleClass(1)

print(example\_object.a)

print(example\_object.b)

The object created by the constructor can have only one of two possible attributes: *a* or *b*. Executing the code will produce the following output:

1

Traceback (most recent call last):

File ".main.py", line 11, in

print(example\_object.b)

AttributeError: 'ExampleClass' object has no attribute 'b'

As you can see, accessing a non-existing object (class) attribute causes an AttributeError exception.

## 3.3.1.7 OOP: Properties: Checking an attribute's existence: continued

The *try-except* instruction gives you the chance to avoid issues with non-existent properties.It's easy - look at the code in the editor.

class ExampleClass:

def \_\_init\_\_(self, val):

if val % 2 != 0:

self.a = 1

else:

self.b = 1

example\_object = ExampleClass(1)

print(example\_object.a)

try:

print(example\_object.b)

except AttributeError:

pass

As you can see, this action isn't very sophisticated. Essentially, we've just swept the issue under the carpet. Fortunately, there is one more way to cope with the issue.

Python provides a function which is able to safely check if any object/class contains a specified property. The function is named *hasattr*, and expects two arguments to be passed to it:

* the class or the object being checked;
* the name of the property whose existence has to be reported (note: it has to be a string containing the attribute name, not the name alone)

The function returns *True* or *False*. This is how you can utilize it:

class ExampleClass:

def \_\_init\_\_(self, val):

if val % 2 != 0:

self.a = 1

else:

self.b = 1

example\_object = ExampleClass(1)

print(example\_object.a)

if hasattr(example\_object, 'b'):

print(example\_object.b)

## 3.3.1.8 OOP: Properties: Checking an attribute's existence: continued

Don't forget that the hasattr() function can operate on classes, too. You can use it to find out if a class variable is available, just like here in the example in the editor.

class ExampleClass:

attr = 1

print(hasattr(ExampleClass, 'attr')) # True

print(hasattr(ExampleClass, 'prop')) # False

The function returns True if the specified class contains a given attribute, and False otherwise. Can you guess the code's output? Run it to check your guesses.

And one more example - look at the code below and try to predict its output:

class ExampleClass:

a = 1

def \_\_init\_\_(self):

self.b = 2

example\_object = ExampleClass()

print(hasattr(example\_object, 'b')) # True

print(hasattr(example\_object, 'a')) # True

print(hasattr(ExampleClass, 'b')) # False

print(hasattr(ExampleClass, 'a')) # True

Were you successful? Run the code to check your predictions.

Okay, we've made it to the end of this section. In the next section we're going to talk about methods, as methods drive the objects and make them active.

## 3.3.1.9 SECTION SUMMARY

**Key takeaways**

1. An **instance variable** is a property whose existence depends on the creation of an object. Every object can have a different set of instance variables.

Moreover, they can be freely added to and removed from objects during their lifetime. All object instance variables are stored inside a dedicated dictionary named \_\_dict\_\_, contained in every object separately.

2. An instance variable can be private when its name starts with \_\_, but don't forget that such a property is still accessible from outside the class using a **mangled name** constructed as \_ClassName\_\_PrivatePropertyName.

3. A **class variable** is a property which exists in exactly one copy, and doesn't need any created object to be accessible. Such variables are not shown as \_\_dict\_\_ content. All a class's class variables are stored inside a dedicated dictionary named \_\_dict\_\_, contained in every class separately.

4. A function named hasattr() can be used to determine if any object/class contains a specified property.

For example:

class Sample:

gamma = 0 # Class variable.

def \_\_init\_\_(self):

self.alpha = 1 # Instance variable.

self.\_\_delta = 3 # Private instance variable.

obj = Sample()

obj.beta = 2 # Another instance variable (existing only inside the "obj" instance.)

print(obj.\_\_dict\_\_)

The code outputs:

{'alpha': 1, '\_Sample\_\_delta': 3, 'beta': 2}

**Exercise 1**：Which of the Python class properties are instance variables and which are class variables? Which of them are private?

class Python:

population = 1

victims = 0

def \_\_init\_\_(self):

self.length\_ft = 3

self.\_\_venomous = False

Check：population and victims are class variables, while length and \_\_venomous are instance variables (the latter is also private).

**Exercise 2**：You're going to negate the \_\_venomous property of the version\_2 object, ignoring the fact that the property is private. How will you do this?

version\_2 = Python()

Check: version\_2.\_Python\_\_venomous = not version\_2.\_Python\_\_venomous

**Exercise 3**: Write an expression which checks if the version\_2 object contains an instance property named constrictor (yes, constrictor!).

Check: hasattr(version\_2, 'constrictor')

# 3.4. METHODS (Class and Object Methods, Constructors, Parameters, Properties)

## **3.4.1.1 OOP: Methods:** Methods in detail

Let's summarize all the facts regarding the use of methods in Python classes. As you already know, a **method is a function embedded inside a class**. There is one fundamental requirement - **a method is obliged to have at least one parameter** (there are no such thing as parameterless methods - a method may be invoked without an argument, but not declared without parameters).

The first (or only) parameter is usually named *self*. We suggest that you follow the convention - it's commonly used, and you'll cause a few surprises by using other names for it. The name *self* suggests the parameter's purpose - it identifies the object for which the method is invoked.

If you're going to invoke a method, you mustn't pass the argument for the self parameter - Python will set it for you. The example in the editor shows the difference.

class Classy:

def method(self):

print("method")

obj = Classy()

obj.method()

The code outputs: method

Note the way we've created the object - we've **treated the class name like a function**, returning a newly instantiated object of the class.

If you want the method to accept parameters other than self, you should:

* place them after self in the method's definition;
* deliver them during invocation without specifying self (as previously)

Just like here:

class Classy:

def method(self, par):

print("method:", par)

obj = Classy()

obj.method(1)

obj.method(2)

obj.method(3)

The code outputs:

method: 1

method: 2

method: 3

## **3.4.1.2 OOP:** Methods in detail: continued 1/9/22

The *self* parameter is used to **obtain access to the object's instance and class variables.** The example shows both ways of utilizing self:

class Classy:

varia = 2

def method(self):

print(self.varia, self.var)

obj = Classy()

obj.var = 3

obj.method()

The code outputs: 2 3

The self parameter is also used to **invoke other object/class methods from inside the class**. Just like here:

class Classy:

def other(self):

print("other")

def method(self):

print("method")

self.other()

obj = Classy()

obj.method()

The code outputs:

method

other

## **3.4.1.3 OOP:** Methods in detail: continued

If you name a method like this: \_\_init\_\_, it won't be a regular method - it will be a **constructor**. If a class has a constructor, it is invoked automatically and implicitly when the object of the class is instantiated. The constructor:

* is **obliged to have the self parameter** (it's set automatically, as usual);
* **may (but doesn't need to) have more parameters** than just self; if this happens, the way in which the class name is used to create the object must reflect the \_\_init\_\_ definition;
* **can be used to set up the object**, i.e., properly initialize its internal state, create instance variables, instantiate any other objects if their existence is needed, etc.

Look at the code in the editor. The example shows a very simple constructor at work.

class Classy:

def \_\_init\_\_(self, value):

self.var = value

obj\_1 = Classy("object")

print(obj\_1.var)

Run it. The code outputs:

object

output

Note that the constructor:

* **cannot return a value**, as it is designed to return a newly created object and nothing else;
* **cannot be invoked directly either from the object or from inside the class** (you can invoke a constructor from any of the object's subclasses, but we'll discuss this issue later.)

## **3.4.1.4 OOP:** Methods in detail: continued

As \_\_init\_\_ is a method, and a method is a function, you can do the same tricks with constructors/methods as you do with ordinary functions. The example in the editor shows how to define a constructor with a default argument value. Test it.

class Classy:

def \_\_init\_\_(self, value = None):

self.var = value

obj\_1 = Classy("object")

obj\_2 = Classy()

print(obj\_1.var)

print(obj\_2.var)

The code outputs:

object

None

Everything we've said about **property name mangling** applies to method names, too - a method whose name starts with \_\_ is (partially) hidden. The example shows this effect:

class Classy:

def visible(self):

print("visible")

def \_\_hidden(self):

print("hidden")

obj = Classy()

obj.visible()

try:

obj.\_\_hidden()

except:

print("failed")

obj.\_Classy\_\_hidden()

The code outputs:

visible

failed

hidden

## **3.4.1.5 OOP:** Methods: The inner life of classes and objects 1/10/22

Each Python class and each Python object is pre-equipped with a set of useful attributes which can be used to examine its capabilities. You already know one of these - it's the \_\_dict\_\_ property. Let's observe how it deals with methods - look at the code in the editor.

class Classy:

varia = 1

def \_\_init\_\_(self):

self.var = 2

def method(self):

pass

def \_\_hidden(self):

pass

obj = Classy()

print(obj.\_\_dict\_\_)

print(Classy.\_\_dict\_\_)

Run it to see what it outputs. Check the output carefully.

{'var': 2}

{'\_\_module\_\_': '\_\_main\_\_', 'varia': 1, '\_\_init\_\_': <function Classy.\_\_init\_\_ at 0x0000025DE184FD30>, 'method': <function Classy.method at 0x0000025DE184FCA0>, '\_Classy\_\_hidden': <function Classy.\_\_hidden at 0x0000025DE184FC10>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Classy' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'Classy' objects>, '\_\_doc\_\_': None}

Find all the defined methods and attributes. Locate the context in which they exist: inside the object or inside the class.

## **3.4.1.6 OOP:** Methods: The inner life of classes and objects: Continued

\_\_dict\_\_ is a dictionary. Another built-in property worth mentioning is \_\_name\_\_, which is a string. The property contains **the name of the class**. It's nothing exciting, just a string.

Note: the \_\_name\_\_ attribute is absent from the object - **it exists only inside classes.**

If you want to **find the class of a particular object**, you can use a function named type(), which is able (among other things) to find a class which has been used to instantiate any object. Look at the code in the editor, run it, and see for yourself.

class Classy:

pass

print(Classy.\_\_name\_\_)

obj = Classy()

print(type(obj).\_\_name\_\_)

The code outputs:

Classy

Classy

Note that a statement like this one: print(obj.\_\_name\_\_)

will cause an error.

## **3.4.1.7 OOP:** Methods: The inner life of classes and objects: Continued

\_\_module\_\_ is a string, too - **it stores the name of the module which contains the definition of the class.** Let's check it - run the code in the editor.

class Classy:

pass

print(Classy.\_\_module\_\_)

obj = Classy()

print(obj.\_\_module\_\_)

The code outputs:

\_\_main\_\_

\_\_main\_\_

As you know, any module named \_\_main\_\_ is actually not a module, but **the file currently being run.**

## **3.4.1.8 OOP:** Methods: The inner life of classes and objects: Continued

\_\_bases\_\_ is a tuple. The **tuple contains classes** (not class names) which are direct superclasses for the class. The order is the same as that used inside the class definition. We'll show you only a very basic example, as we want to highlight **how inheritance works.** Moreover, we're going to show you how to use this attribute when we discuss the objective aspects of exceptions.

Note: **only classes have this attribute** - objects don't.

We've defined a function named *printbases*(), designed to present the tuple's contents clearly. Look at the code in the editor. Analyze it and run it.

class SuperOne:

pass

class SuperTwo:

pass

class Sub(SuperOne, SuperTwo):

pass

def printBases(cls):

print('( ', end='')

for x in cls.\_\_bases\_\_:

print(x.\_\_name\_\_, end=' ')

print(')')

printBases(SuperOne)

printBases(SuperTwo)

printBases(Sub)

It will output:

( object )

( object )

( SuperOne SuperTwo )

Note: **a class without explicit superclasses points to object** (a predefined Python class) as its direct ancestor.

## **3.4.1.8 OOP:** Methods: Reflection and introspection

All these means allow the Python programmer to perform two important activities specific to many objective languages. They are:

* **introspection**, which is the ability of a program to examine the type or properties of an object at runtime;
* **reflection**, which goes a step further, and is the ability of a program to manipulate the values, properties and/or functions of an object at runtime.

In other words, you don't have to know a complete class/object definition to manipulate the object, as the object and/or its class contain the metadata allowing you to recognize its features during program execution.

## **3.4.1.10 OOP:** Methods: Investigating classes

What can you find out about classes in Python? The answer is simple - everything. Both reflection and introspection enable a programmer to do anything with every object, no matter where it comes from. Analyze the the code in the editor.

class MyClass:

pass

obj = MyClass()

obj.a = 1

obj.b = 2

obj.i = 3

obj.ireal = 3.5

obj.integer = 4

obj.z = 5

def incIntsI(obj):

for name in obj.\_\_dict\_\_.keys():

if name.startswith('i'):

val = getattr(obj, name)

if isinstance(val, int):

setattr(obj, name, val + 1)

print(obj.\_\_dict\_\_)

incIntsI(obj)

print(obj.\_\_dict\_\_)

The function named incIntsI() gets an object of any class, scans its contents in order to find all integer attributes with names starting with i, and increments them by one. Impossible? Not at all! This is how it works:

* line 1: define a very simple class...
* lines 3 through 10: ... and fill it with some attributes;
* line 12: this is our function!
* line 13: scan the \_\_dict\_\_ attribute, looking for all attribute names;
* line 14: if a name starts with i...
* line 15: ... use the getattr() function to get its current value; note: getattr() takes two arguments: an object, and its property name (as a string), and returns the current attribute's value;
* line 16: check if the value is of type integer, and use the function isinstance() for this purpose (we'll discuss this later);
* line 17: if the check goes well, increment the property's value by making use of the setattr() function; the function takes three arguments: an object, the property name (as a string), and the property's new value.

The code outputs:

{'a': 1, 'integer': 4, 'b': 2, 'i': 3, 'z': 5, 'ireal': 3.5}

{'a': 1, 'integer': 5, 'b': 2, 'i': 4, 'z': 5, 'ireal': 3.5}

That's all!

## **3.4.1.11** SECTION SUMMARY

**Key takeaways**

1. A method is a function embedded inside a class. The first (or only) parameter of each method is usually named self, which is designed to identify the object for which the method is invoked in order to access the object's properties or invoke its methods.

2. If a class contains a **constructor** (a method named \_\_init\_\_) it cannot return any value and cannot be invoked directly.

3. All classes (but not objects) contain a property named \_\_name\_\_, which stores the name of the class. Additionally, a property named \_\_module\_\_ stores the name of the module in which the class has been declared, while the property named \_\_bases\_\_ is a tuple containing a class's superclasses. For example:

class Sample:

def \_\_init\_\_(self):

self.name = Sample.\_\_name\_\_

def myself(self):

print("My name is " + self.name + " living in a " + Sample.\_\_module\_\_)

obj = Sample()

obj.myself()

The code outputs:

My name is Sample living in a \_\_main\_\_

## **3.4.1.12 The Timer class (To be worked on) 1/11/22**

## **3.4.1.13 Days of the week (To be worked on)**

## **3.4.1.14 Points on a plane (To be worked on)**

## **3.4.1.15 Triangle (To be worked on)**

# 3.5. INHERITANCE (Functions, Methods, Class Hierarchies, Polymorphism, Composition, Single vs Multiple Inheritance)

## **3.5.1.1 OOP Fundamentals:** Inheritance - why and how?

Before we start talking about inheritance, we want to present a new, handy mechanism utilized by Python's classes and objects - **it's the way in which the object is able to introduce itself.** Let's start with an example. Look at the code in the editor.

class Star:

def \_\_init\_\_(self, name, galaxy):

self.name = name

self.galaxy = galaxy

sun = Star("Sun", "Milky Way")

print(sun)

The program prints out just one line of text, which in our case is this:

<\_\_main\_\_.Star object at 0x7f1074cc7c50>

If you run the same code on your computer, you'll see something very similar, although the hexadecimal number (the substring starting with 0x) will be different, as it's just an internal object identifier used by Python, and it's unlikely that it would appear the same when the same code is run in a different environment. As you can see, the printout here isn't really useful, and something more specific, or just prettier, may be more preferable. Fortunately, Python offers just such a function.

## **3.5.1.2 OOP Fundamentals:** Inheritance - why and how?

When Python needs any class/object to be presented as a string (putting an object as an argument in the print() function invocation fits this condition) it tries to invoke a method named \_\_str\_\_() from the object and to use the string it returns.

The default \_\_str\_\_() method returns the previous string - ugly and not very informative. You can change it just by **defining your own method of the name**. We've just done it - look at the code in the editor.

class Star:

def \_\_init\_\_(self, name, galaxy):

self.name = name

self.galaxy = galaxy

def \_\_str\_\_(self):

return self.name + ' in ' + self.galaxy

sun = Star("Sun", "Milky Way")

print(sun)

This new \_\_str\_\_() method makes a string consisting of the star's and galaxy's names - nothing special, but the print results look better now, doesn't it?Can you guess the output? Run the code to check if you were right. Output:Sun in Milky Way

## **3.5.1.3 OOP Fundamentals:** Inheritance - why and how?

The term inheritance is older than computer programming, and it describes the common practice of passing different goods from one person to another upon that person's death. The term, when related to computer programming, has an entirely different meaning. Let's define the term for our purposes:

Inheritance is a common practice (in object programming) of **passing attributes and methods from the superclass (defined and existing) to a newly created class, called the subclass.** In other words, **inheritance is a way of building a new class, not from scratch, but by using an already defined repertoire of traits.** The new class inherits (and this is the key) all the already existing equipment, but is able to add some new ones if needed. Thanks to that, it's possible to **build more specialized (more concrete) classes** using some sets of predefined general rules and behaviors.

The most important factor of the process is the relation between the superclass and all of its subclasses (note: if B is a subclass of A and C is a subclass of B, this also means than C is a subclass of A, as the relationship is fully transitive). A very simple example of **two-level inheritance** is presented here:

class Vehicle:

pass

class LandVehicle(Vehicle):

pass

class TrackedVehicle(LandVehicle):

pass

All the presented classes are empty for now, as we're going to show you how the mutual relations between the super- and subclasses work. We'll fill them with contents soon. We can say that:

* The Vehicle class is the superclass for LandVehicle and TrackedVehicle classes;
* The LandVehicle class is a subclass of Vehicle and a superclass of TrackedVehicle at the same time;
* The TrackedVehicle class is a subclass of Vehicle and LandVehicle classes.

The above knowledge comes from reading the code (in other words, we know it because we can see it). Does Python know the same? Is it possible to ask Python about it? Yes, it is.

## **3.5.1.4 OOP Fundamentals:** Inheritance: issubclass()

Python offers a function which is able to **identify a relationship between two classes,** and although its diagnosis isn't complex, it can **check if a particular class is a subclass of any other class.** This is how it looks:

issubclass(ClassOne, ClassTwo)

The function returns True if ClassOne is a subclass of ClassTwo, and False otherwise.

Let's see it in action - it may surprise you. Look at the code in the editor:

class Vehicle:

pass

class LandVehicle(Vehicle):

pass

class TrackedVehicle(LandVehicle):

pass

for cls1 in [Vehicle, LandVehicle, TrackedVehicle]:

for cls2 in [Vehicle, LandVehicle, TrackedVehicle]:

print(issubclass(cls1, cls2), end="\t")

print()

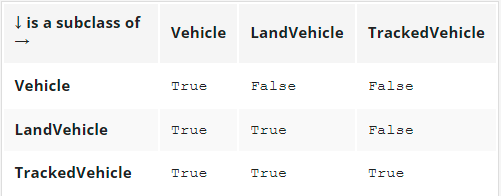
There are two nested loops. Their purpose is to **check all possible ordered pairs of classes, and to print the results of the check to determine whether the pair matches the subclass-superclass relationship.** Run the code. The program produces the following output:

True False False

True True False

True True True

Let's make the result more readable:



There is one important observation to make: **each class is considered to be a subclass of itself.**

## **3.5.1.5 OOP Fundamentals:** Inheritance: isinstance()

As you already know, **an object is an incarnation(化身) of a class.** This means that the object is like a cake baked using a recipe which is included inside the class. This can generate some important issues.

Let's assume that you've got a cake (e.g., as an argument passed to your function). You want to know what recipe has been used to make it. Why? Because you want to know what to expect from it, e.g., whether it contains nuts or not, which is crucial information to some people.

Similarly, it can be crucial if the object does have (or doesn't have) certain characteristics. In other words, **whether it is an object of a certain class or not.** Such a fact could be detected by the function named isinstance():

isinstance(objectName, ClassName)

The functions returns True if the object is an instance of the class, or False otherwise.

**Being an instance of a class means that the object (the cake) has been prepared using a recipe contained in either the class or one of its superclasses.**

Don't forget: if a subclass contains at least the same equipment as any of its superclasses, it means that objects of the subclass can do the same as objects derived from the superclass, ergo, it's an instance of its home class and any of its superclasses.

Let's test it. Analyze the code in the editor.

class Vehicle:

pass

class LandVehicle(Vehicle):

pass

class TrackedVehicle(LandVehicle):

pass

my\_vehicle = Vehicle()

my\_land\_vehicle = LandVehicle()

my\_tracked\_vehicle = TrackedVehicle()

for obj in [my\_vehicle, my\_land\_vehicle, my\_tracked\_vehicle]:

for cls in [Vehicle, LandVehicle, TrackedVehicle]:

print(isinstance(obj, cls), end="\t")

print()

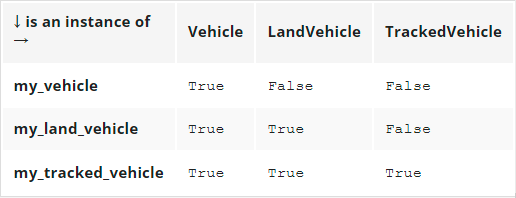
We've created three objects, one for each of the classes. Next, using two nested loops, we check all possible object-class pairs to **find out if the objects are instances of the classes.** Run the code.This is what we get:

True False False

True True False

True True True

Let's make the result more readable once again:



Does the table confirm our expectations?

## **3.5.1.6 OOP Fundamentals:** Inheritance: the is operator

There is also a Python operator worth mentioning, as it refers directly to objects - here it is:

object\_one is object\_two

**The *is* operator checks whether two variables (object\_one and object\_two here) refer to the same object.**

Don't forget that **variables don't store the objects themselves, but only the handles pointing to the internal Python memory.**

Assigning a value of an object variable to another variable doesn't copy the object, but only its handle. This is why an operator like *is* may be very useful in particular circumstances. Take a look at the code in the editor.

class SampleClass:

def \_\_init\_\_(self, val):

self.val = val

object\_1 = SampleClass(0)

object\_2 = SampleClass(2)

object\_3 = object\_1

object\_3.val += 1

print(object\_1 is object\_2)

print(object\_2 is object\_3)

print(object\_3 is object\_1)

print(object\_1.val, object\_2.val, object\_3.val)

string\_1 = "Mary had a little "

string\_2 = "Mary had a little lamb"

string\_1 += "lamb"

print(string\_1 == string\_2, string\_1 is string\_2)

Let's analyze it:

* there is a very simple class equipped with a simple constructor, creating just one property. The class is used to instantiate two objects. The former is then assigned to another variable, and its val property is incremented by one.
* afterward, the is operator is applied three times to check all possible pairs of objects, and all val property values are also printed.
* the last part of the code carries out another experiment. After three assignments, both strings contain the same texts, but these texts are stored in different objects.

The code prints:

False

False

True

1 2 1

True False

The results prove that object\_1 and object\_3 are actually the same objects, while string\_1 and string\_2 aren't, despite their contents being the same.

## **3.5.1.7 How Python finds properties and methods** 1/12/22

Now we're going to look at how Python deals with inheriting methods. Take a look at the example in the editor.

class Super:

def \_\_init\_\_(self, name):

self.name = name

def \_\_str\_\_(self):

return "My name is " + self.name + "."

class Sub(Super):

def \_\_init\_\_(self, name):

Super.\_\_init\_\_(self, name)

obj = Sub("Andy")

print(obj)

Let's analyze it:

* there is a class named Super, which defines its own constructor used to assign the object's property, named name.
* the class defines the \_\_str\_\_() method, too, which makes the class able to present its identity in clear text form.
* the class is next used as a base to create a subclass named Sub. The Sub class defines its own constructor, which invokes the one from the superclass. Note how we've done it: Super.\_\_init\_\_(self, name).
* we've explicitly named the superclass, and pointed to the method to invoke \_\_init\_\_(), providing all needed arguments.
* we've instantiated one object of class Sub and printed it.

The code outputs: My name is Andy.

Note: As there is no \_\_str\_\_() method within the Sub class, the printed string is to be produced within the Super class. This means that the \_\_str\_\_() method has been inherited by the Sub class.

## **3.5.1.8 How Python finds properties and methods: continued**

Look at the code in the editor. We've modified it to show you another method of accessing any entity defined inside the superclass.

class Super:

def \_\_init\_\_(self, name):

self.name = name

def \_\_str\_\_(self):

return "My name is " + self.name + "."

class Sub(Super):

def \_\_init\_\_(self, name):

super().\_\_init\_\_(name)

obj = Sub("Andy")

print(obj)

In the last example, we explicitly named the superclass. In this example, we make use of the super() function, which **accesses the superclass without needing to know its name:** super().\_\_init\_\_(name)

The super() function creates a context in which you don't have to (moreover, you mustn't) pass the self argument to the method being invoked - this is why it's possible to activate the superclass constructor using only one argument.

Note: you can use this mechanism not only to **invoke the superclass constructor, but also to get access to any of the resources available inside the superclass.**

## **3.5.1.9 How Python finds properties and methods: continued**

Let's try to do something similar, but with properties (more precisely: with class variables). Take a look at the example in the editor.

# Testing properties: class variables.

class Super:

supVar = 1

class Sub(Super):

subVar = 2

obj = Sub()

print(obj.subVar)

print(obj.supVar)

As you can see, the Super class defines one class variable named supVar, and the Sub class defines a variable named subVar.

Both these variables are visible inside the object of class Sub - this is why the code outputs:

2

1

## **3.5.1.10 How Python finds properties and methods: continued**

The same effect can be observed with instance variables - take a look at the second example in the editor.

# Testing properties: instance variables.

class Super:

def \_\_init\_\_(self):

self.supVar = 11

class Sub(Super):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.subVar = 12

obj = Sub()

print(obj.subVar)

print(obj.supVar)

The Sub class constructor creates an instance variable named subVar, while the Super constructor does the same with a variable named supVar. As previously, both variables are accessible from within the object of class Sub. The program's output is:

12

11

Note: the existence of the supVar variable is obviously conditioned by the Super class constructor invocation. Omitting it would result in the absence of the variable in the created object (try it yourself).

## **3.5.1.11 How Python finds properties and methods: continued**

It's now possible to formulate a general statement describing Python's behavior. When you try to access any object's entity, Python will try to (in this order):

* find it **inside the object** itself;
* find it **in all classes** involved in the object's inheritance line from bottom to top;

If both of the above fail, an **exception (AttributeError) is raised.**

The first condition may need some additional attention. As you know, all objects deriving from a particular class may have different sets of attributes, and some of the attributes may be added to the object a long time after the object's creation.

The example in the editor summarizes this in a **three-level inheritance line.** Analyze it carefully.

class Level1:

variable\_1 = 100

def \_\_init\_\_(self):

self.var\_1 = 101

def fun\_1(self):

return 102

class Level2(Level1):

variable\_2 = 200

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.var\_2 = 201

def fun\_2(self):

return 202

class Level3(Level2):

variable\_3 = 300

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.var\_3 = 301

def fun\_3(self):

return 302

obj = Level3()

print(obj.variable\_1, obj.var\_1, obj.fun\_1()) # 100 101 102

print(obj.variable\_2, obj.var\_2, obj.fun\_2()) # 200 201 202

print(obj.variable\_3, obj.var\_3, obj.fun\_3()) # 300 301 302

All the comments we've made so far are related to **single inheritance**, when a subclass has exactly one superclass. This is the most common situation (and the recommended one, too).

Python, however, offers much more here. In the next lessons we're going to show you some examples of **multiple inheritance.**

## **3.5.1.12 How Python finds properties and methods: continued**

**Multiple inheritance occurs when a class has more than one superclass.** Syntactically, such inheritance is presented as a comma-separated list of superclasses put inside parentheses after the new class name - just like here:

class SuperA:

var\_a = 10

def fun\_a(self):

return 11

class SuperB:

var\_b = 20

def fun\_b(self):

return 21

class Sub(SuperA, SuperB):

pass

obj = Sub()

print(obj.var\_a, obj.fun\_a())

print(obj.var\_b, obj.fun\_b())

The Sub class has two superclasses: SuperA and SuperB. This means that the Sub class **inherits all the goods offered by both SuperA and SuperB**. The code prints:

10 11

20 21

Now it's time to introduce a brand new term - **overriding**. What do you think will happen if more than one of the superclasses defines an entity of a particular name?

## **3.5.1.13 How Python finds properties and methods: continued**

Let's analyze the example in the editor.

class Level1:

var = 100

def fun(self):

return 101

class Level2(Level1):

var = 200

def fun(self):

return 201

class Level3(Level2):

pass

obj = Level3()

print(obj.var, obj.fun())

Both, Level1 and Level2 classes define a method named fun() and a property named var. Does this mean that the Level3 class object will be able to access two copies of each entity? Not at all.

**The entity defined later (in the inheritance sense) overrides the same entity defined earlier.** This is why the code produces the following output: 200 201

As you can see, the var class variable and fun() method from the Level2 class override the entities of the same names derived from the Level1 class.

This feature can be intentionally used to modify default (or previously defined) class behaviors when any of its classes needs to act in a different way to its ancestor.

We can also say that **Python looks for an entity from bottom to top**, and is fully satisfied with the first entity of the desired name.

How does it work when a class has two ancestors offering the same entity, and they lie on the same level? In other words, what should you expect when a class emerges using multiple inheritance? Let's look at this.

## **3.5.1.14 How Python finds properties and methods: continued**

Let's take a look at the example in the editor.

class Left:

var = "L"

var\_left = "LL"

def fun(self):

return "Left"

class Right:

var = "R"

var\_right = "RR"

def fun(self):

return "Right"

class Sub(Left, Right):

pass

obj = Sub()

print(obj.var, obj.var\_left, obj.var\_right, obj.fun())

The Sub class inherits goods from two superclasses, Left and Right (these names are intended to be meaningful). There is no doubt that the class variable var\_right comes from the Right class, and var\_left comes from Left respectively.

This is clear. But where does var come from? Is it possible to guess it? The same problem is encountered with the fun() method - will it be invoked from Left or from Right? Let's run the program - its output is: L LL RR Left

This proves that both unclear cases have a solution inside the Left class. Is this a sufficient premise to formulate a general rule? Yes, it is.

We can say that Python looks for **object components** in the following order:

* inside the object itself;
* in its superclasses, from bottom to top;
* if there is more than one class on a particular inheritance path, Python scans them from left to right.

Do you need anything more? Just make a small amendment in the code - replace: class Sub(Left, Right): with: class Sub(Right, Left):, then run the program again, and see what happens. What do you see now? We see: R LL RR Right

Do you see the same, or something different?

## **3.5.1.15** How to build a hierarchy of classes

Building a hierarchy of classes isn't just art for art's sake. If you divide a problem among classes and decide which of them should be located at the top and which should be placed at the bottom of the hierarchy, you have to carefully analyze the issue, but before we show you how to do it (and how not to do it), we want to highlight an interesting effect. It's nothing extraordinary (it's just a consequence of the general rules presented earlier), but remembering it may be key to understanding how some codes work, and how the effect may be used to build a flexible set of classes.

Take a look at the code in the editor.

class One:

def do\_it(self):

print("do\_it from One")

def doanything(self):

self.do\_it()

class Two(One):

def do\_it(self):

print("do\_it from Two")

one = One()

two = Two()

one.doanything()

two.doanything()

Let's analyze it:

* there are two classes, named One and Two, while Two is derived from One. Nothing special. However, one thing looks remarkable - the do\_it() method.
* the do\_it()method is **defined twice**: originally inside One and subsequently inside Two. The essence of the example lies in the fact that it is **invoked just once** - inside One.

The question is - which of the two methods will be invoked by the last two lines of the code?

The first invocation seems to be simple, and it is simple, actually - invoking doanything() from the object named one will obviously activate the first of the methods.

The second invocation needs some attention. It's simple, too if you keep in mind how Python finds class components. The second invocation will launch do\_it() in the form existing inside the Two class, regardless of the fact that the invocation takes place within the One class. In effect, the code produces the following output:

do\_it from One

do\_it from Two

Note: the situation in which the **subclass is able to modify its superclass behavior (just like in the example) is called polymorphism.** The word comes from Greek (polys: "many, much" and morphe, "form, shape"), which means that one and the same class can take various forms depending on the redefinitions done by any of its subclasses.

The method, redefined in any of the superclasses, thus changing the behavior of the superclass, is called **virtual**. In other words, no class is given once and for all. Each class's behavior may be modified at any time by any of its subclasses. We're going to show you **how to use polymorphism to extend class flexibility.**

## **3.5.1.16** How to build a hierarchy of classes: continued

Look at the example in the editor.

import time

class TrackedVehicle:

def control\_track(left, stop):

pass

def turn(left):

control\_track(left, True)

time.sleep(0.25)

control\_track(left, False)

class WheeledVehicle:

def turn\_front\_wheels(left, on):

pass

def turn(left):

turn\_front\_wheels(left, True)

time.sleep(0.25)

turn\_front\_wheels(left, False)

Does it resemble anything? Yes, of course it does. It refers to the example shown at the beginning of the module when we talked about the general concepts of objective programming. It may look weird, but we didn't use inheritance in any way - just to show you that it doesn't limit us, and we managed to get ours.

We defined two separate classes able to produce two different kinds of land vehicles. The main difference between them is in how they turn. A wheeled vehicle just turns the front wheels (generally). A tracked vehicle has to stop one of the tracks.

Can you follow the code?

* a tracked vehicle performs a turn by stopping and moving on one of its tracks (this is done by the control\_track() method, which will be implemented later)
* a wheeled vehicle turns when its front wheels turn (this is done by the turn\_front\_wheels() method)
* the turn() method uses the method suitable for each particular vehicle.

Can you see **what's wrong with the code**? The turn() methods look too similar to leave them in this form.

Let's rebuild the code - we're going to introduce a superclass to gather all the similar aspects of the driving vehicles, moving all the specifics to the subclasses.

## **3.5.1.17** How to build a hierarchy of classes: continued 1/13/22

Look at the code in the editor again.

import time

class Vehicle:

def change\_direction(left, on):

pass

def turn(left):

change\_direction(left, True)

time.sleep(0.25)

change\_direction(left, False)

class TrackedVehicle(Vehicle):

def control\_track(left, stop):

pass

def change\_direction(left, on):

control\_track(left, on)

class WheeledVehicle(Vehicle):

def turn\_front\_wheels(left, on):

pass

def change\_direction(left, on):

turn\_front\_wheels(left, on)

pass

This is what we've done:

* we defined a superclass named Vehicle, which uses the turn() method to implement a general scheme of turning, while the turning itself is done by a method named change\_direction(); note: the former method is empty, as we are going to put all the details into the subclass (such a method is often called an **abstract method**, as it only demonstrates some possibility which will be instantiated later)
* we defined a subclass named TrackedVehicle (note: it's derived from the Vehicle class) which instantiated the change\_direction() method by using the specific (concrete) method named control\_track()
* respectively, the subclass named WheeledVehicle does the same trick, but uses the turn\_front\_wheels() method to force the vehicle to turn.

The most important advantage (omitting readability issues) is that this form of code enables you to implement a brand new turning algorithm just by modifying the turn() method, which can be done in just one place, as all the vehicles will obey it. This is how **polymorphism helps the developer to keep the code clean and consistent.**

## **3.5.1.18** How to build a hierarchy of classes: continued

Inheritance is not the only way of constructing adaptable classes. You can achieve the same goals (not always, but very often) by using a technique named composition.

**Composition is the process of composing an object using other different objects.** The objects used in the composition deliver a set of desired traits (properties and/or methods) so we can say that they act like blocks used to build a more complicated structure.

It can be said that:

* **inheritance extends a class's capabilities** by adding new components and modifying existing ones; in other words, the complete recipe is contained inside the class itself and all its ancestors; the object takes all the class's belongings and makes use of them;
* **composition projects a class as a container** able to store and use other objects (derived from other classes) where each of the objects implements a part of a desired class's behavior.

Let us illustrate the difference by using the previously defined vehicles. The previous approach led us to a hierarchy of classes in which the top-most class was aware of the general rules used in turning the vehicle, but didn't know how to control the appropriate components (wheels or tracks).

The subclasses implemented this ability by introducing specialized mechanisms. Let's do (almost) the same thing, but using composition. The class - like in the previous example - is aware of how to turn the vehicle, but the actual turn is done by a specialized object stored in a property named controller. The controller is able to control the vehicle by manipulating the relevant vehicle's parts. Take a look into the editor - this is how it could look.

import time

class Tracks:

def change\_direction(self, left, on):

print("tracks: ", left, on)

class Wheels:

def change\_direction(self, left, on):

print("wheels: ", left, on)

class Vehicle:

def \_\_init\_\_(self, controller):

self.controller = controller

def turn(self, left):

self.controller.change\_direction(left, True)

time.sleep(0.25)

self.controller.change\_direction(left, False)

wheeled = Vehicle(Wheels())

tracked = Vehicle(Tracks())

wheeled.turn(True)

tracked.turn(False)

There are two classes named Tracks and Wheels - they know how to control the vehicle's direction. There is also a class named Vehicle which can use any of the available controllers (the two already defined, or any other defined in the future) - the controller itself is passed to the class during initialization.

In this way, the vehicle's ability to turn is composed using an external object, not implemented inside the Vehicle class. In other words, we have a universal vehicle and can install either tracks or wheels onto it. The code produces the following output:

wheels: True True

wheels: True False

tracks: False True

tracks: False False

## **3.5.1.19** Single inheritance vs. multiple inheritance

As you already know, there are no obstacles to using multiple inheritance in Python. You can derive any new class from more than one previously defined classes. There is only one "but". The fact that you can do it does not mean you have to.

Don't forget that:

* a single inheritance class is always simpler, safer, and easier to understand and maintain;
* multiple inheritance is always risky, as you have many more opportunities to make a mistake in identifying these parts of the superclasses which will effectively influence the new class;
* multiple inheritance may make overriding extremely tricky; moreover, using the super() function becomes ambiguous;
* multiple inheritance violates the **single responsibility principle** (more details here: https://en.wikipedia.org/wiki/Single\_responsibility\_principle) as it makes a new class of two (or more) classes that know nothing about each other;
* we strongly suggest multiple inheritance as the last of all possible solutions - if you really need the many different functionalities offered by different classes, composition may be a better alternative.

## **3.5.1.20** OOP Fundamentals: MRO

**What is Method Resolution Order (MRO) and why is it that not all inheritances make sense?**

MRO, in general, is a way (you can call it a **strategy**) in which a particular programming language scans through the upper part of a class’s hierarchy in order to find the method it currently needs. It's worth emphasizing that different languages use slightly (or even completely) different MROs. Python is a unique creature in this respect, however, and its customs are a bit specific.

We're going to show you how Python's MRO works in two peculiar cases that are clear-cut examples of problems which may occur when you try to use multiple inheritance too recklessly. Let's start with a snippet that initially may look simple. Look at what we've prepared for you in the editor.

class Top:

def m\_top(self):

print("top")

class Middle(Top):

def m\_middle(self):

print("middle")

class Bottom(Middle):

def m\_bottom(self):

print("bottom")

object = Bottom()

object.m\_bottom()

object.m\_middle()

object.m\_top()

We're sure that if you analyze the snippet yourself, you won't see any anomalies in it. Yes, you're perfectly right - it looks clear and simple, and raises no concerns. If you run the code, it will produce the following, predictable output:

bottom

middle

top

No surprises so far. Let's make a tiny change to this code. Have a look:

class Top:

def m\_top(self):

print("top")

class Middle(Top):

def m\_middle(self):

print("middle")

class Bottom(Middle, Top):

def m\_bottom(self):

print("bottom")

object = Bottom()

object.m\_bottom()

object.m\_middle()

object.m\_top()

Can you see the difference? It's hidden in this line: class Bottom(Middle, Top):

In this exotic way, we've turned a very simple code with a clear single-inheritance path into a mysterious multiple-inheritance riddle. “Is it valid?” you may ask. Yes, it is. “How is that possible?” you should ask now, and we hope that you really feel the need to ask this question.

As you can see, the order in which the two superclasses have been listed between parenthesis is compliant with the code's structure: the Middle class precedes the Top class, just like in the real inheritance path.

Despite its oddity, the sample is correct and works as expected, but it has to be stated that this notation doesn’t bring any new functionality or additional meaning.

Let's modify the code once again - now we'll swap both superclass names in the Bottom class definition. This is what the snippet looks like now:

class Top:

def m\_top(self):

print("top")

class Middle(Top):

def m\_middle(self):

print("middle")

class Bottom(Top, Middle):

def m\_bottom(self):

print("bottom")

object = Bottom()

object.m\_bottom()

object.m\_middle()

object.m\_top()

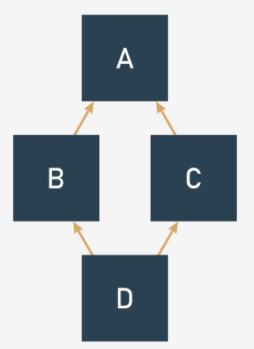
To anticipate your question, we’ll say that this amendment has spoiled the code, and it won't run anymore. What a pity. The order we tried to force (Top, Middle) is incompatible with the inheritance path which is derived from the code's structure. Python won't like it. This is what we'll see:

*TypeError: Cannot create a consistent method resolution order (MRO) for bases Top, Middle*

We think that the message speaks for itself. Python's MRO cannot be bent or violated, not just because that's the way Python works, but also because it’s a rule you have to obey.

## **3.5.1.21** OOP Fundamentals: MRO: The diamond problem

The second example of the spectrum of issues that can possibly arise from multiple inheritance is illustrated by a classic problem named the **diamond problem**. The name reflects the shape of the inheritance diagram - take a look at the picture:



* There is the top-most superclass named A;
* there are two subclasses derived from A: B and C;
* and there is also the bottom-most subclass named D, derived from B and C (or C and B, as these two variants mean different things in Python)

Can you see the diamond there?

Have a look at the code in the editor. The same structure, but expressed in Python.

class A:

pass

class B(A):

pass

class C(A):

pass

class D(B, C):

pass

d = D()

Some programming languages forbid multiple inheritance at all, and as a consequence, they won't let you build a diamond - this is the route that Java and C# have chosen to follow since their origins.

Python, however, has chosen a different route - it allows multiple inheritance, and it doesn't mind if you write and run code like the one in the editor. But don't forget about MRO - it's always in charge. Let's rebuild our example from the previous page to make it more diamond-like, just like below:

class Top:

def m\_top(self):

print("top")

class Middle\_Left(Top):

def m\_middle(self):

print("middle\_left")

class Middle\_Right(Top):

def m\_middle(self):

print("middle\_right")

class Bottom(Middle\_Left, Middle\_Right):

def m\_bottom(self):

print("bottom")

object = Bottom()

object.m\_bottom()

object.m\_middle()

object.m\_top()

Note: both Middle classes define **a method of the same name**: m\_middle().

It introduces a small uncertainty to our sample, although we're absolutely sure that you can answer the following key question: which of the two m\_middle() methods will actually be invoked when the following line is executed?

Object.m\_middle()

In other words, what will you see on the screen: middle\_left or middle\_right? You don't need to hurry – think twice and keep Python's MRO in mind! Are you ready?

Yes, you're right. The invocation will activate the m\_middle() method, which comes from the Middle\_Left class. The explanation is simple: the class is listed before Middle\_Right on the Bottom class's inheritance list. If you want to make sure there’s no doubt about it, try to swap these two classes on the list and check the results.

If you want to experience some more profound impressions about multiple inheritance and precious gemstones, try to modify our snippet and equip the Upper class with another specimen of the m\_middle() method, and investigate its behavior carefully. As you can see, diamonds may bring some problems into your life – both the real ones and those offered by Python.

## **3.5.1.22 SECTION SUMMARY 1/2**

**Key takeaways**

1. A method named \_\_str\_\_() is responsible for **converting an object's contents into a (more or less) readable string.** You can redefine it if you want your object to be able to present itself in a more elegant form. For example:

class Mouse:

def \_\_init\_\_(self, name):

self.my\_name = name

def \_\_str\_\_(self):

return self.my\_name

the\_mouse = Mouse('mickey')

print(the\_mouse) # Prints "mickey".

2. A function named issubclass(Class\_1, Class\_2) is able to determine if Class\_1 is a **subclass** of Class\_2. For example:

class Mouse:

pass

class LabMouse(Mouse):

pass

print(issubclass(Mouse, LabMouse), issubclass(LabMouse, Mouse))

# Prints "False True"

3. A function named isinstance(Object, Class) checks if an object **comes from an indicated class.** For example:

class Mouse:

pass

class LabMouse(Mouse):

pass

mickey = Mouse()

print(isinstance(mickey, Mouse), isinstance(mickey, LabMouse))

# Prints "True False".

4. A operator called *is* checks if two variables refer to **the same object**. For example:

class Mouse:

pass

mickey = Mouse()

minnie = Mouse()

cloned\_mickey = mickey

print(mickey is minnie, mickey is cloned\_mickey) # Prints "False True".

5. A parameterless function named super() returns a **reference to the nearest superclass of the class.** For example:

class Mouse:

def \_\_str\_\_(self):

return "Mouse"

class LabMouse(Mouse):

def \_\_str\_\_(self):

return "Laboratory " + super().\_\_str\_\_()

doctor\_mouse = LabMouse();

print(doctor\_mouse) # Prints "Laboratory Mouse".

6. Methods as well as instance and class variables defined in a superclass are **automatically inherited** by their subclasses. For example:

class Mouse:

Population = 0

def \_\_init\_\_(self, name):

Mouse.Population += 1

self.name = name

def \_\_str\_\_(self):

return "Hi, my name is " + self.name

class LabMouse(Mouse):

pass

professor\_mouse = LabMouse("Professor Mouser")

print(professor\_mouse, Mouse.Population)

# Prints "Hi, my name is Professor Mouser 1"

7. In order to find any object/class property, Python looks for it inside:

* the object itself;
* all classes involved in the object's inheritance line from bottom to top;
* if there is more than one class on a particular inheritance path, Python scans them from left to right;
* if both of the above fail, the AttributeError exception is raised.

8. If any of the subclasses defines a method/class variable/instance variable of the same name as existing in the superclass, the new name **overrides** any of the previous instances of the name. For example:

class Mouse:

def \_\_init\_\_(self, name):

self.name = name

def \_\_str\_\_(self):

return "My name is " + self.name

class AncientMouse(Mouse):

def \_\_str\_\_(self):

return "Meum nomen est " + self.name

mus = AncientMouse("Caesar")

print(mus) # Prints "Meum nomen est Caesar"

## **3.5.1.23 SECTION SUMMARY 2/2**

**Exercises**

Scenario: Assume that the following piece of code has been successfully executed:

class Dog:

kennel = 0

def \_\_init\_\_(self, breed):

self.breed = breed

Dog.kennel += 1

def \_\_str\_\_(self):

return self.breed + " says: Woof!"

class SheepDog(Dog):

def \_\_str\_\_(self):

return super().\_\_str\_\_() + " Don't run away, Little Lamb!"

class GuardDog(Dog):

def \_\_str\_\_(self):

return super().\_\_str\_\_() + " Stay where you are, Mister Intruder!"

rocky = SheepDog("Collie")

luna = GuardDog("Dobermann")

Now answer the questions from exercises 1-4.

**Exercise 1: What is the expected output of the following piece of code?**

print(rocky)

print(luna)

Check：Collie says: Woof! Don't run away, Little Lamb!

Dobermann says: Woof! Stay where you are, Mister Intruder!

**Exercise 2:** **What is the expected output of the following piece of code?**

print(issubclass(SheepDog, Dog), issubclass(SheepDog, GuardDog))

print(isinstance(rocky, GuardDog), isinstance(luna, GuardDog))

Check：True False

False True

**Exercise 3: What is the expected output of the following piece of code?**

print(luna is luna, rocky is luna)

print(rocky.kennel)

Check：True False

2

**Exercise 4：Define a SheepDog's subclass named LowlandDog, and equip it with an \_\_str\_\_() method overriding an inherited method of the same name. The new dog's \_\_str\_\_() method should return the string "Woof! I don't like mountains!" .**

Check

class LowlandDog(SheepDog):

def \_\_str\_\_(self):

return Dog.\_\_str\_\_(self) + " I don't like mountains!"

# 3.6. THE OBJECTIVE NATURE OF PYTHON EXCEPTIONS

## 3.6.1.1 Exceptions once again: More about exceptions 1/14/22

Discussing object programming offers a very good opportunity to return to exceptions. The objective nature of Python's exceptions makes them a very flexible tool, able to fit to specific needs, even those you don't yet know about.

Before we dive into the **objective face of exceptions**, we want to show you some syntactical and semantic aspects of the way in which Python treats the try-except block, as it offers a little more than what we have presented so far.

The first feature we want discuss here is an additional, possible branch that can be placed inside (or rather, directly behind) the try-except block - it's the part of the code starting with else - just like in the example in the editor.

def reciprocal(n):

try:

n = 1 / n

except ZeroDivisionError:

print("Division failed")

return None

else:

print("Everything went fine")

return n

print(reciprocal(2))

print(reciprocal(0))

A code labelled in this way is executed when (and only when) no exception has been raised inside the try: part. We can say that exactly one branch can be executed after try: - either the one beginning with except (don't forget that there can be more than one branch of this kind) or the one starting with else.

Note: the else: branch has to be located after the last except branch.

The example code produces the following output:

Everything went fine

0.5

Division failed

None

## 3.6.1.2 Exceptions once again: More about exceptions

The try-except block can be extended in one more way - by adding a part headed by the *finally* keyword (it must be the last branch of the code designed to handle exceptions). Note: these two variants (*else* and *finally*) aren't dependent in any way, and they can coexist or occur independently.

The finally block is always executed (it finalizes the try-except block execution, hence its name), no matter what happened earlier, even when raising an exception, no matter whether this has been handled or not.

def reciprocal(n):

try:

n = 1 / n

except ZeroDivisionError:

print("Division failed")

n = None

else:

print("Everything went fine")

finally:

print("It's time to say goodbye")

return n

print(reciprocal(2))

print(reciprocal(0))

Look at the code in the editor. It outputs:

Everything went fine

It's time to say good bye

0.5

Division failed

It's time to say good bye

None

## 3.6.1.3 Exceptions once again:Exceptions are classes

All the previous examples were content with detecting a specific kind of exception and responding to it in an appropriate way. Now we're going to delve deeper, and look inside the exception itself.

You probably won't be surprised to learn that **exceptions are classes**. Furthermore, when an exception is raised, an object of the class is instantiated, and goes through all levels of program execution, looking for the except branch that is prepared to deal with it.

Such an object carries some useful information which can help you to precisely identify all aspects of the pending situation. To achieve that goal, Python offers a special variant of the exception clause - you can find it in the editor.

try:

i = int("Hello!")

except Exception as e:

print(e)

print(e.\_\_str\_\_())

As you can see, the except statement is extended, and contains an additional phrase starting with the as keyword, followed by an identifier. The identifier is designed to catch the exception object so you can analyze its nature and draw proper conclusions.

Note: the identifier's scope covers its except branch, and doesn't go any further.

The example presents a very simple way of utilizing the received object - just print it out (as you can see, the output is produced by the object's \_\_str\_\_() method) and it contains a brief message describing the reason.

*invalid literal for int() with base 10: 'Hello!'*

*invalid literal for int() with base 10: 'Hello!'*

The same message will be printed if there is no fitting except block in the code, and Python is forced to handle it alone.

## 3.6.1.4 Exceptions once again: Exceptions are classes

All the built-in Python exceptions form a hierarchy of classes. There is no obstacle to extending it if you find it reasonable. Look at the code in the editor.

def print\_exception\_tree(thisclass, nest = 0):

if nest > 1:

print(" |" \* (nest - 1), end="")

if nest > 0:

print(" +---", end="")

print(thisclass.\_\_name\_\_)

for subclass in thisclass.\_\_subclasses\_\_():

print\_exception\_tree(subclass, nest + 1)

print\_exception\_tree(BaseException)

This program dumps all predefined exception classes in the form of a tree-like printout.

As **a tree is a perfect example of a recursive data structure**, a recursion seems to be the best tool to traverse through it. The print\_exception\_tree() function takes two arguments:

* a point inside the tree from which we start traversing the tree;
* a nesting level (we'll use it to build a simplified drawing of the tree's branches)

Let's start from the tree's root - the root of Python's exception classes is the BaseException class (it's a superclass of all other exceptions). For each of the encountered classes, perform the same set of operations:

* print its name, taken from the \_\_name\_\_ property;
* iterate through the list of subclasses delivered by the \_\_subclasses\_\_() method, and recursively invoke the print\_exception\_tree() function, incrementing the nesting level respectively.

Note how we've drawn the branches and forks. The printout isn't sorted in any way - you can try to sort it yourself, if you want a challenge. Moreover, there are some subtle inaccuracies in the way in which some branches are presented. That can be fixed, too, if you wish. This is how it looks:

BaseException

+---Exception

| +---TypeError

| +---StopAsyncIteration

| +---StopIteration

| +---ImportError

| | +---ModuleNotFoundError

| | +---ZipImportError

| +---OSError

| | +---ConnectionError

| | | +---BrokenPipeError

| | | +---ConnectionAbortedError

| | | +---ConnectionRefusedError

| | | +---ConnectionResetError

| | +---BlockingIOError

| | +---ChildProcessError

| | +---FileExistsError

| | +---FileNotFoundError

| | +---IsADirectoryError

| | +---NotADirectoryError

| | +---InterruptedError

| | +---PermissionError

| | +---ProcessLookupError

| | +---TimeoutError

| | +---UnsupportedOperation

| | +---herror

| | +---gaierror

| | +---timeout

| | +---Error

| | | +---SameFileError

| | +---SpecialFileError

| | +---ExecError

| | +---ReadError

| +---EOFError

| +---RuntimeError

| | +---RecursionError

| | +---NotImplementedError

| | +---\_DeadlockError

| | +---BrokenBarrierError

| +---NameError

| | +---UnboundLocalError

| +---AttributeError

| +---SyntaxError

| | +---IndentationError

| | | +---TabError

| +---LookupError

| | +---IndexError

| | +---KeyError

| | +---CodecRegistryError

| +---ValueError

| | +---UnicodeError

| | | +---UnicodeEncodeError

| | | +---UnicodeDecodeError

| | | +---UnicodeTranslateError

| | +---UnsupportedOperation

| +---AssertionError

| +---ArithmeticError

| | +---FloatingPointError

| | +---OverflowError

| | +---ZeroDivisionError

| +---SystemError

| | +---CodecRegistryError

| +---ReferenceError

| +---BufferError

| +---MemoryError

| +---Warning

| | +---UserWarning

| | +---DeprecationWarning

| | +---PendingDeprecationWarning

| | +---SyntaxWarning

| | +---RuntimeWarning

| | +---FutureWarning

| | +---ImportWarning

| | +---UnicodeWarning

| | +---BytesWarning

| | +---ResourceWarning

| +---error

| +---Verbose

| +---Error

| +---TokenError

| +---StopTokenizing

| +---Empty

| +---Full

| +---\_OptionError

| +---TclError

| +---SubprocessError

| | +---CalledProcessError

| | +---TimeoutExpired

| +---Error

| | +---NoSectionError

| | +---DuplicateSectionError

| | +---DuplicateOptionError

| | +---NoOptionError

| | +---InterpolationError

| | | +---InterpolationMissingOptionError

| | | +---InterpolationSyntaxError

| | | +---InterpolationDepthError

| | +---ParsingError

| | | +---MissingSectionHeaderError

| +---InvalidConfigType

| +---InvalidConfigSet

| +---InvalidFgBg

| +---InvalidTheme

| +---EndOfBlock

| +---BdbQuit

| +---error

| +---\_Stop

| +---PickleError

| | +---PicklingError

| | +---UnpicklingError

| +---\_GiveupOnSendfile

| +---error

| +---LZMAError

| +---RegistryError

| +---ErrorDuringImport

+---GeneratorExit

+---SystemExit

+---KeyboardInterrupt

## 3.6.1.5 Exceptions once again: Detailed anatomy of exceptions

Let's take a closer look at the exception's object, as there are some really interesting elements here (we'll return to the issue soon when we consider Python's input/output base techniques, as their exception subsystem extends these objects a bit).

The *BaseException* class introduces a property named *args*. It's a **tuple designed to gather all arguments passed to the class constructor.** It is empty if the construct has been invoked without any arguments, or contains just one element when the constructor gets one argument (we don't count the self argument here), and so on.

We've prepared a simple function to print the args property in an elegant way. You can see the function in the editor.

def print\_args(args):

lng = len(args)

if lng == 0:

print("")

elif lng == 1:

print(args[0])

else:

print(str(args))

try:

raise Exception

except Exception as e:

print(e, e.\_\_str\_\_(), sep=' : ' ,end=' : ')

print\_args(e.args)

try:

raise Exception("my exception")

except Exception as e:

print(e, e.\_\_str\_\_(), sep=' : ', end=' : ')

print\_args(e.args)

try:

raise Exception("my", "exception")

except Exception as e:

print(e, e.\_\_str\_\_(), sep=' : ', end=' : ')

print\_args(e.args)

We've used the function to print the contents of the *args* property in three different cases, where the exception of the Exception class is raised in three different ways. To make it more spectacular, we've also printed the object itself, along with the result of the \_\_str\_\_() invocation.

The first case looks routine - there is just the name Exception after the raise keyword. This means that the object of this class has been created in a most routine way.

The second and third cases may look a bit weird at first glance, but there's nothing odd here - these are just the constructor invocations. In the second raise statement, the constructor is invoked with one argument, and in the third, with two.

As you can see, the program output reflects this, showing the appropriate contents of the args property:

: :

my exception : my exception : my exception

('my', 'exception') : ('my', 'exception') : ('my', 'exception')

## 3.6.1.6 How to create your own exception

The exceptions hierarchy is neither closed nor finished, and you can always extend it if you want or need to create your own world populated with your own exceptions.

It may be useful when you create a complex module which detects errors and raises exceptions, and you want the exceptions to be easily distinguishable from any others brought by Python. This is done by **defining your own, new exceptions as subclasses derived from predefined ones.**

Note: if you want to create an exception which will be utilized as a specialized case of any built-in exception, derive it from just this one. If you want to build your own hierarchy, and don't want it to be closely connected to Python's exception tree, derive it from any of the top exception classes, like Exception.

Imagine that you've created a brand new arithmetic, ruled by your own laws and theorems. It's clear that division has been redefined, too, and has to behave in a different way than routine dividing. It's also clear that this new division should raise its own exception, different from the built-in ZeroDivisionError, but it's reasonable to assume that in some circumstances, you (or your arithmetic's user) may want to treat all zero divisions in the same way.

Demands like these may be fulfilled in the way presented in the editor. Look at the code, and let's analyze it:

* We've defined our own exception, named MyZeroDivisionError, derived from the built-in ZeroDivisionError. As you can see, we've decided not to add any new components to the class.

In effect, an exception of this class can be - depending on the desired point of view - treated like a plain ZeroDivisionError, or considered separately.

* The do\_the\_division() function raises either a MyZeroDivisionError or ZeroDivisionError exception, depending on the argument's value.

The function is invoked four times in total, while the first two invocations are handled using only one except branch (the more general one) and the last two ones with two different branches, able to distinguish the exceptions (don't forget: the order of the branches makes a fundamental difference!)

## 3.6.1.7 How to create your own exception:continued

When you're going to build a completely new universe filled with completely new creatures that have nothing in common with all the familiar things, you may want to **build your own exception structure.** For example, if you work on a large simulation system which is intended to model the activities of a pizza restaurant, it can be desirable to form a separate hierarchy of exceptions.

You can start building it by **defining a general exception as a new base class** for any other specialized exception. We've done in in the following way:

class PizzaError(Exception):

def \_\_init\_\_(self, pizza, message):

Exception.\_\_init\_\_(self, message)

self.pizza = pizza

Note: we're going to collect more specific information here than a regular Exception does, so our constructor will take two arguments:

* one specifying a pizza as a subject of the process,
* and one containing a more or less precise description of the problem.

As you can see, we pass the second parameter to the superclass constructor, and save the first inside our own property.

A more specific problem (like an excess of cheese) can require a more specific exception. It's possible to derive the new class from the already defined PizzaError class, like we've done here:

class TooMuchCheeseError(PizzaError):

def \_\_init\_\_(self, pizza, cheese, message):

PizzaError.\_init\_\_(self, pizza, message)

self.cheese = cheese

The TooMuchCheeseError exception needs more information than the regular PizzaError exception, so we add it to the constructor - the name cheese is then stored for further processing.

## 3.6.1.8 How to create your own exception:continued 1/15/22

Look at the code in the editor. We've coupled together the two previously defined exceptions and harnessed them to work in a small example snippet.

class PizzaError(Exception):

def \_\_init\_\_(self, pizza, message):

Exception.\_\_init\_\_(self, message)

self.pizza = pizza

class TooMuchCheeseError(PizzaError):

def \_\_init\_\_(self, pizza, cheese, message):

PizzaError.\_\_init\_\_(self, pizza, message)

self.cheese = cheese

def make\_pizza(pizza, cheese):

if pizza not in ['margherita', 'capricciosa', 'calzone']:

raise PizzaError(pizza, "no such pizza on the menu")

if cheese > 100:

raise TooMuchCheeseError(pizza, cheese, "too much cheese")

print("Pizza ready!")

for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:

try:

make\_pizza(pz, ch)

except TooMuchCheeseError as tmce:

print(tmce, ':', tmce.cheese)

except PizzaError as pe:

print(pe, ':', pe.pizza)

One of these is raised inside the make\_pizza() function when any of these two erroneous situations is discovered: a wrong pizza request, or a request for too much cheese. Note:

* removing the branch starting with except TooMuchCheeseError will cause all appearing exceptions to be classified as PizzaError;
* removing the branch starting with except PizzaErrorwill cause the TooMuchCheeseError exceptions to remain unhandled, and will cause the program to terminate.

The previous solution, although elegant and efficient, has one important weakness. Due to the somewhat easygoing way of declaring the constructors, the new exceptions cannot be used as-is, without a full list of required arguments.

We'll remove this weakness by **setting the default values for all constructor parameters**. Take a look:

class PizzaError(Exception):

def \_\_init\_\_(self, pizza='uknown', message=''):

Exception.\_\_init\_\_(self, message)

self.pizza = pizza

class TooMuchCheeseError(PizzaError):

def \_\_init\_\_(self, pizza='uknown', cheese='>100', message=''):

PizzaError.\_\_init\_\_(self, pizza, message)

self.cheese = cheese

def make\_pizza(pizza, cheese):

if pizza not in ['margherita', 'capricciosa', 'calzone']:

raise PizzaError

if cheese > 100:

raise TooMuchCheeseError

print("Pizza ready!")

for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:

try:

make\_pizza(pz, ch)

except TooMuchCheeseError as tmce:

print(tmce, ':', tmce.cheese)

except PizzaError as pe:

print(pe, ':', pe.pizza)

Now, if the circumstances permit, it is possible to use the class names alone.

## 3.6.1.9 SECTION SUMMARY: **Key takeaways**

1. The else: branch of the try statement is executed when there has been no exception during the execution of the try: block.

2. The finally: branch of the try statement is **always** executed.

3. The syntax *except Exception\_Name as an exception\_object*: lets you intercept an object carrying information about a pending exception. The object's property named args (a tuple) stores all arguments passed to the object's constructor.

4. The exception classes can be extended to enrich them with new capabilities, or to adopt their traits to newly defined exceptions. For example:

try:

assert \_\_name\_\_ == "\_\_main\_\_"

except:

print("fail", end=' ')

else:

print("success", end=' ')

finally:

print("done")

The code outputs: success done.

**Exercise 1: What is the expected output of the following code?**

import math

try:

print(math.sqrt(9))

except ValueError:

print("inf")

else:

print("fine")

Check

3.0

fine

**Exercise 2: What is the expected output of the following code?**

import math

try:

print(math.sqrt(-9))

except ValueError:

print("inf")

else:

print("fine")

finally:

print("the end")

Check

inf

the end

**Exercise 3: What is the expected output of the following code?**

class NewValueError(ValueError):

def \_\_init\_\_(self, name, color, state):

self.data = (name, color, state)

try:

raise NewValueError("Enemy warning", "Red alert", "High readiness")

except NewValueError as nve:

for arg in nve.args:

print(arg, end='! ')

Check: Enemy warning! Red alert! High readiness!

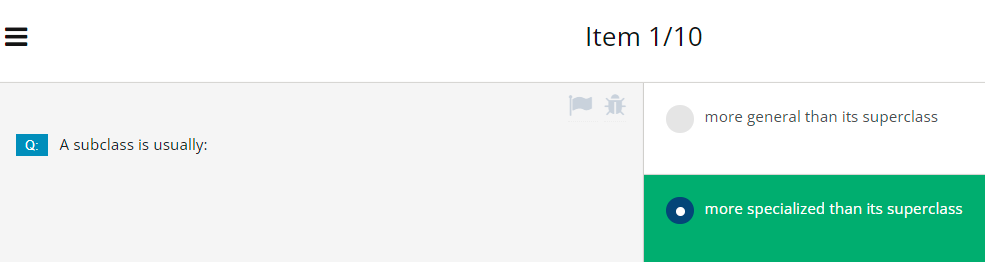
## 3.6.1.10 Module Completion

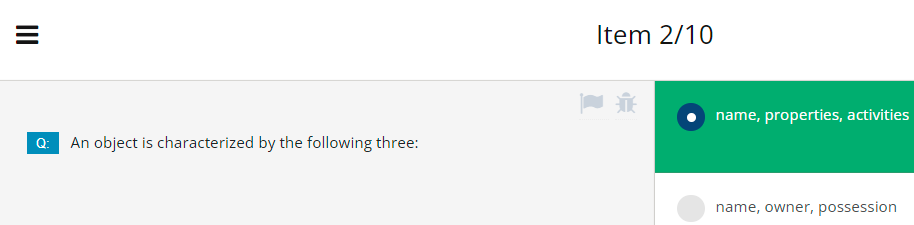
Well done! You've reached the end of Module 3 and completed a major milestone in your Python programming education. Here's a short summary of the objectives you've covered and got familiar with in Module 3:

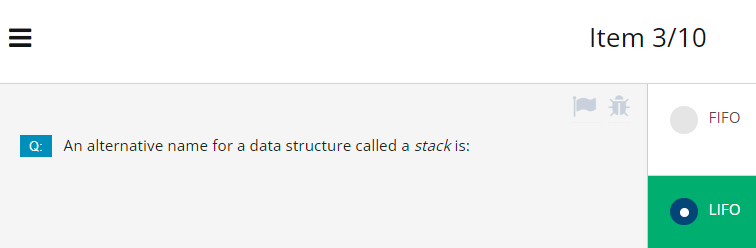
* the foundations and basic concepts of object-oriented programming;
* the differences between the procedural and object approaches on the example of the stack;
* properties (instance and class variables, attributes)
* methods (class and object methods, the constructor, parameters, and properties)
* the concept of inheritance (functions, methods, class hierarchies, polymorphism, composition, single vs. multiple inheritance)
* the objective nature of Python exceptions.

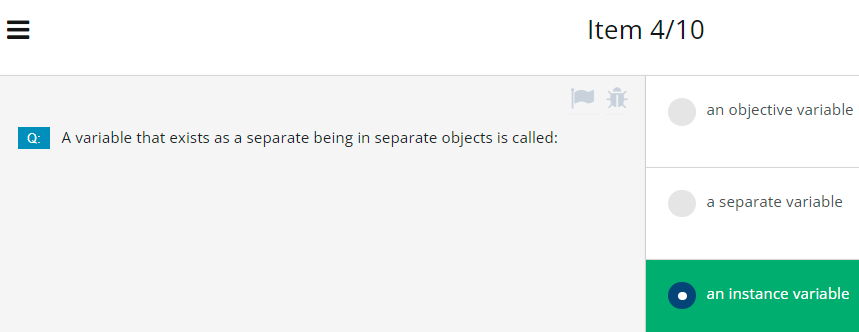
You are now ready to take the module quiz and attempt the final challenge: Module 3 Test, which will help you gauge what you've learned so far.

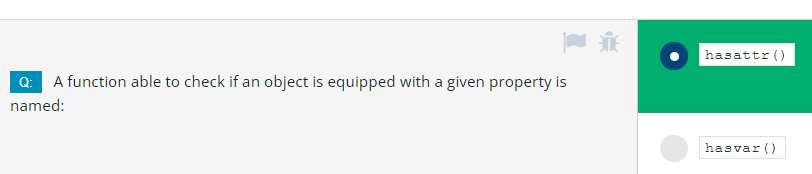
# 3.7 PE2 -- Module 3 Quiz

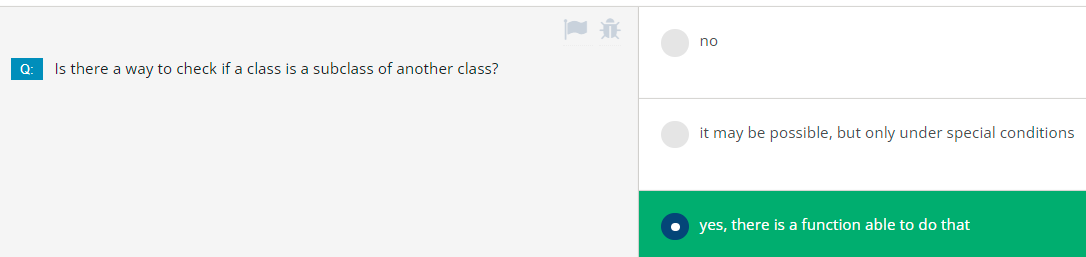


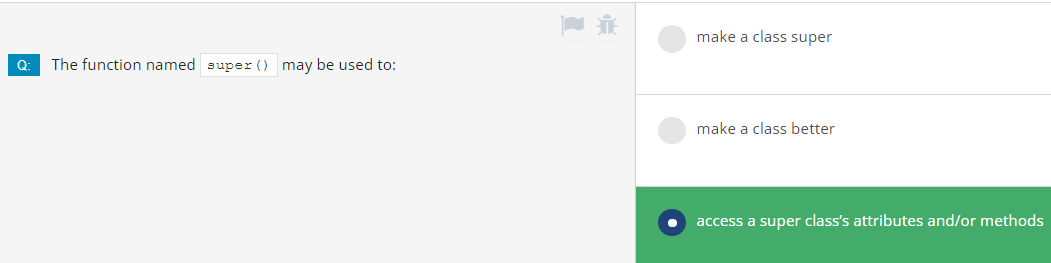


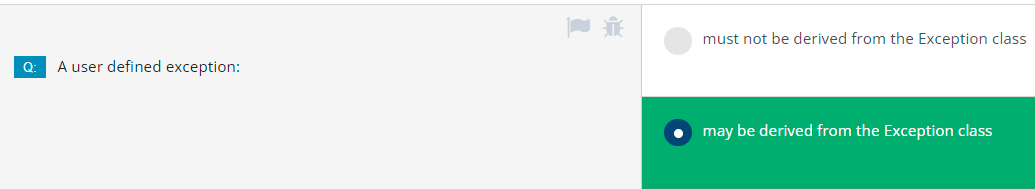


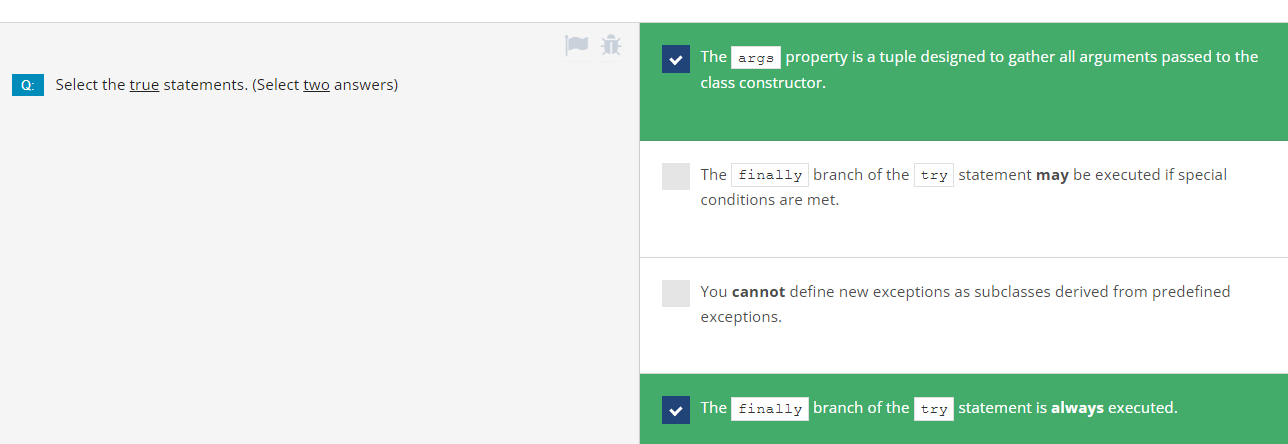


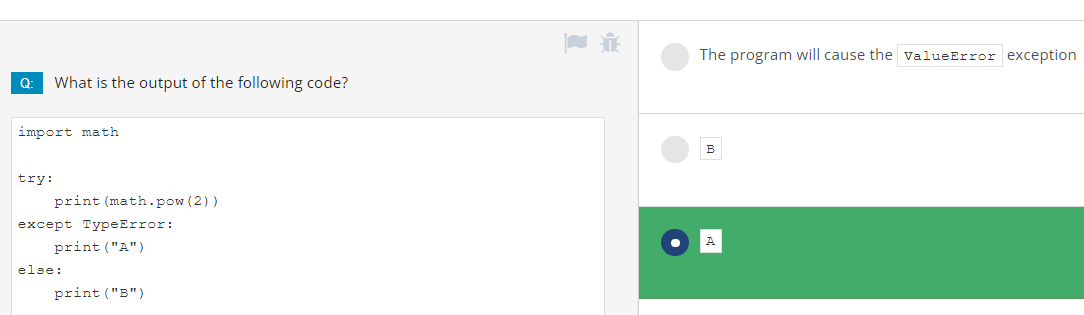




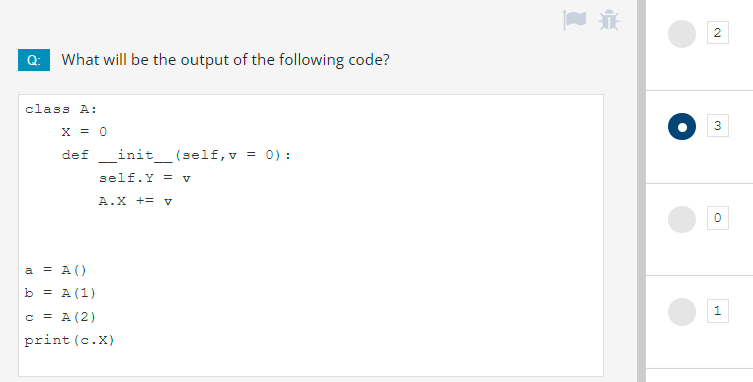


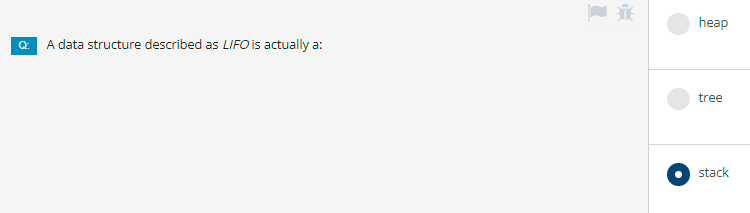
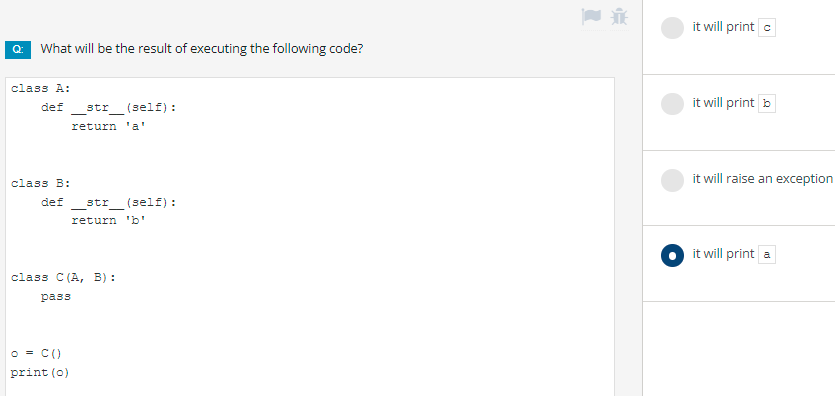


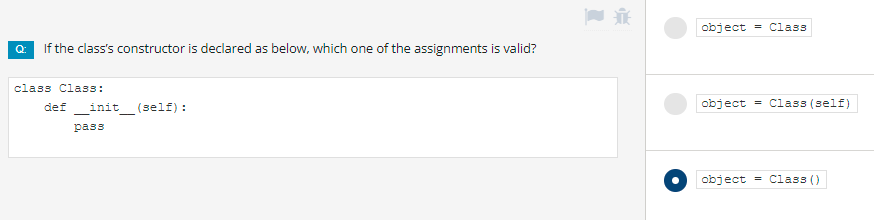


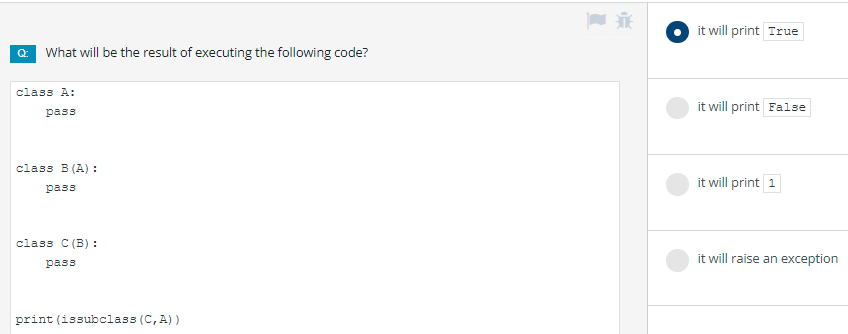


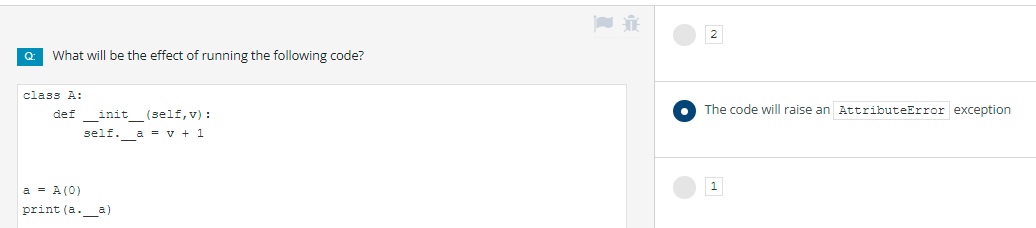
# Module 3 Test

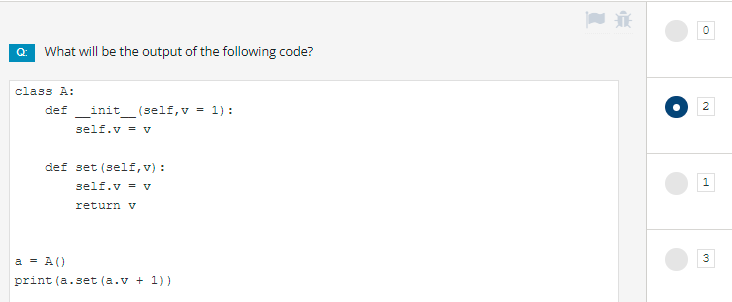


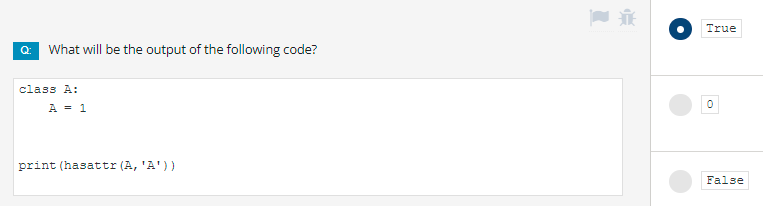


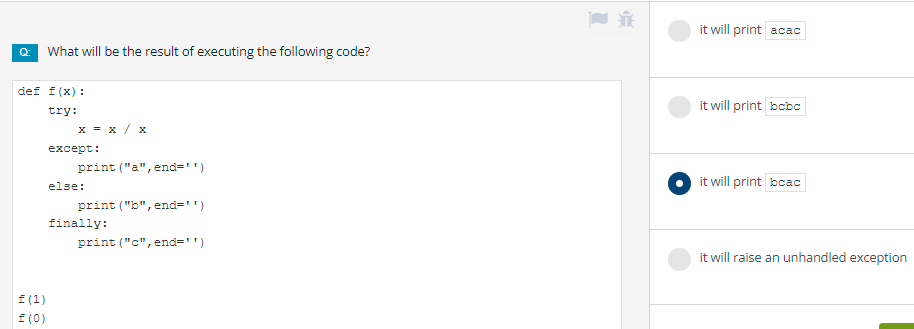


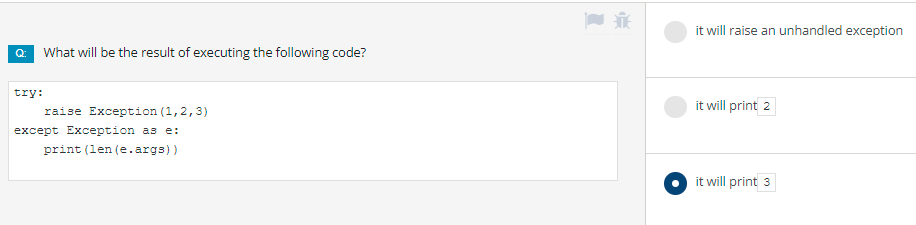












# **MODULE 4: MISCELLANEOUS**

# **4.1. GENERATORS, ITERATORS, AND CLOSURES**

## 4.1.1.0 Python Essentials 2 - Module 4

Python Essentials 2: Module 4Miscellaneous

In this module, you will learn about:

* Generators, iterators and closures;
* Working with file-system, directory tree and files;
* Selected Python Standard Library modules (os, datetime, time, and calendar.)

## 4.1.1.1 Generators and closures: **Generators - where to find them**

**Generator** - what do you associate this word with? Perhaps it refers to some electronic device. Or perhaps it refers to a heavy and serious machine designed to produce power, electrical or other.

A Python generator is **a piece of specialized code able to produce a series of values, and to control the iteration process.** This is why generators are very often called **iterators**, and although some may find a very subtle distinction between these two, we'll treat them as one. You may not realize it, but you've encountered generators many, many times before. Take a look at the very simple snippet:

for i in range(5):

print(i)

The range() function is, in fact, a generator, which is (in fact, again) an iterator. What is the difference? A function returns one, well-defined value - it may be the result of a more or less complex evaluation of, e.g., a polynomial, and is invoked once - only once. A generator **returns a series of values**, and in general, is (implicitly) invoked more than once.

In the example, the range() generator is invoked six times, providing five subsequent values from zero to four, and finally signaling that the series is complete.

The above process is completely transparent. Let's shed some light on it. Let's show you the **iterator protocol**.

## 4.1.1.2 Generators and closures: Generators - where to find them: continued

**The iterator protocol is a way in which an object should behave to conform to the rules imposed by the context of the *for* and *in* statements**. An object conforming to the iterator protocol is called an **iterator**.

An iterator must provide two methods:

* \_\_iter\_\_() which should **return the object itself** and which is invoked once (it's needed for Python to successfully start the iteration)
* \_\_next\_\_() which is intended to **return the next value** (first, second, and so on) of the desired series - it will be invoked by the for/in statements in order to pass through the next iteration; if there are no more values to provide, the method should raise the StopIteration exception.

Does it sound strange? Not at all. Look at the example in the editor.

class Fib:

def \_\_init\_\_(self, nn):

print("\_\_init\_\_")

self.\_\_n = nn

self.\_\_i = 0

self.\_\_p1 = self.\_\_p2 = 1

def \_\_iter\_\_(self):

print("\_\_iter\_\_")

return self

def \_\_next\_\_(self):

print("\_\_next\_\_")

self.\_\_i += 1

if self.\_\_i > self.\_\_n:

raise StopIteration

if self.\_\_i in [1, 2]:

return 1

ret = self.\_\_p1 + self.\_\_p2

self.\_\_p1, self.\_\_p2 = self.\_\_p2, ret

return ret

for i in Fib(10):

print(i)

We've built a class able to iterate through the first n values (where n is a constructor parameter) of the Fibonacci numbers. Let us remind you - the Fibonacci numbers (Fibi) are defined as follows:

Fib1 = 1

Fib2 = 1

Fibi = Fibi-1 + Fibi-2

In other words:

* the first two Fibonacci numbers are equal to 1;
* any other Fibonacci number is the sum of the two previous ones (e.g., Fib3 = 2, Fib4 = 3, Fib5 = 5, and so on)

Let's dive into the code:

* lines 2 through 6: the class constructor prints a message (we'll use this to trace the class's behavior), prepares some variables (\_\_n to store the series limit, \_\_i to track the current Fibonacci number to provide, and \_\_p1 along with \_\_p2 to save the two previous numbers);
* lines 8 through 10: the \_\_iter\_\_ method is obliged to return the iterator object itself; its purpose may be a bit ambiguous here, but there's no mystery; try to imagine an object which is not an iterator (e.g., it's a collection of some entities), but one of its components is an iterator able to scan the collection; the \_\_iter\_\_ method should **extract the iterator and entrust it with the execution of the iteration protocol**; as you can see, the method starts its action by printing a message;
* lines 12 through 21: the \_\_next\_\_ method is responsible for creating the sequence; it's somewhat wordy, but this should make it more readable; first, it prints a message, then it updates the number of desired values, and if it reaches the end of the sequence, the method breaks the iteration by raising the StopIteration exception; the rest of the code is simple, and it precisely reflects the definition we showed you earlier;
* lines 24 and 25 make use of the iterator.

The code produces the following output:

\_\_init\_\_

\_\_iter\_\_

\_\_next\_\_

1

\_\_next\_\_

1

\_\_next\_\_

2

\_\_next\_\_

3

\_\_next\_\_

5

\_\_next\_\_

8

\_\_next\_\_

13

\_\_next\_\_

21

\_\_next\_\_

34

\_\_next\_\_

55

\_\_next\_\_

Look:

* the iterator object is instantiated first;
* next, Python invokes the \_\_iter\_\_ method to get access to the actual iterator;
* the \_\_next\_\_ method is invoked eleven times - the first ten times produce useful values, while the eleventh terminates the iteration.

## 4.1.1.3 Generators and closures: Generators - where to find them: continued

The previous example shows you a solution where the iterator **object is a part of a more complex class.** The code isn't really sophisticated, but it presents the concept in a clear way. Take a look at the code in the editor.

class Fib:

def \_\_init\_\_(self, nn):

self.\_\_n = nn

self.\_\_i = 0

self.\_\_p1 = self.\_\_p2 = 1

def \_\_iter\_\_(self):

print("Fib iter")

return self

def \_\_next\_\_(self):

self.\_\_i += 1

if self.\_\_i > self.\_\_n:

raise StopIteration

if self.\_\_i in [1, 2]:

return 1

ret = self.\_\_p1 + self.\_\_p2

self.\_\_p1, self.\_\_p2 = self.\_\_p2, ret

return ret

class Class:

def \_\_init\_\_(self, n):

self.\_\_iter = Fib(n)

def \_\_iter\_\_(self):

print("Class iter")

return self.\_\_iter;

object = Class(8)

for i in object:

print(i)

We've built the Fib iterator into another class (we can say that we've composed it into the Class class). It's instantiated along with Class's object.

The object of the class may be used as an iterator when (and only when) it positively answers to the \_\_iter\_\_ invocation - this class can do it, and if it's invoked in this way, it provides an object able to obey the iteration protocol.

This is why the output of the code is the same as previously, although the object of the Fib class isn't used explicitly inside the for loop's context.

Class iter

1

1

2

3

5

8

13

21

## 4.1.1.4 Generators and closures: The yield statement

The iterator protocol isn't particularly difficult to understand and use, but it is also indisputable that the **protocol is rather inconvenient**. The main discomfort it brings is **the need to save the state of the iteration between subsequent \_\_iter\_\_ invocations.** For example, the Fib iterator is forced to precisely store the place in which the last invocation has been stopped (i.e., the evaluated number and the values of the two previous elements). This makes the code larger and less comprehensible.

This is why Python offers a much more effective, convenient, and elegant way of writing iterators. The concept is fundamentally based on a very specific and powerful mechanism provided by the *yield* keyword. You may think of the *yield* keyword as a smarter sibling of the *return* statement, with one essential difference. Take a look at this function:

def fun(n):

for i in range(n):

return i

It looks strange, doesn't it? It's clear that the *for* loop has no chance to finish its first execution, as the *return* will break it irrevocably. Moreover, invoking the function won't change anything - the *for* loop will start from scratch and will be broken immediately. We can say that such a function is not able to save and restore its state between subsequent invocations. This also means that a function like this **cannot be used as a generator.** We've replaced exactly one word in the code - can you see it?

def fun(n):

for i in range(n):

yield i

We've added *yield* instead of *return*. This little amendment **turns the function into a generator**, and executing the *yield* statement has some very interesting effects.

First of all, it provides the value of the expression specified after the *yield* keyword, just like *return*, but doesn't lose the state of the function. All the variables' values are frozen, and wait for the next invocation, when the execution is resumed (not taken from scratch, like after *return*).

There is one important limitation: such **a function should not be invoked explicitly** as - in fact - it isn't a function anymore; **it's a generator object.** The invocation will **return the object's identifier**, not the series we expect from the generator. Due to the same reasons, the previous function (the one with the return statement) may only be invoked explicitly, and must not be used as a generator.

**How to build a generator**

Let us show you the new generator in action. This is how we can use it:

def fun(n):

for i in range(n):

yield i

for v in fun(5):

print(v)

Can you guess the output?

0

1

2

3

4

## 4.1.1.5 Generators and closures: How to build your own generator

What if you need a **generator to produce the first n powers of 2?** Nothing easier. Just look at the code below:

def powers\_of\_2(n):

power = 1

for i in range(n):

yield power

power \*= 2

for v in powers\_of\_2(8):

print(v)

output

1

2

4

8

16

32

64

128

**List comprehensions**

Generators may also be used within **list comprehensions**, just like here:

def powers\_of\_2(n):

power = 1

for i in range(n):

yield power

power \*= 2

t = [x for x in powers\_of\_2(5)]

print(t)

Run the example and check the output: [1, 2, 4, 8, 16]

**The list() function**

The list() function can transform a series of subsequent generator invocations into **a real list**:

def powers\_of\_2(n):

power = 1

for i in range(n):

yield power

power \*= 2

t = list(powers\_of\_2(3))

print(t)

Again, try to predict the output and run the code to check your predictions: [1, 2, 4]

**The in operator**

Moreover, the context created by the *in* operator allows you to use a generator, too.

The example shows how to do it:

def powers\_of\_2(n):

power = 1

for i in range(n):

yield power

power \*= 2

for i in range(20):

if i in powers\_of\_2(4):

print(i)

What's the code's output? Run the program and check.

1

2

4

8

**The Fibanacci number generator**

Now let's see a **Fibonacci number generator**, and ensure that it looks much better than the objective version based on the direct iterator protocol implementation.

Here it is:

def fibonacci(n):

p = pp = 1

for i in range(n):

if i in [0, 1]:

yield 1

else:

n = p + pp

pp, p = p, n

yield n

fibs = list(fibonacci(10))

print(fibs)

Guess the output (a list) produced by the generator, and run the code to check if you were right. [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

## 4.1.1.6 Generators and closures: More about list comprehensions

You should be able to remember the rules governing the creation and use of a very special Python phenomenon named **list comprehension - a simple and very impressive way of creating lists and their contents.** In case you need it, we've provided a quick reminder in the editor.

list\_1 = []

for ex in range(6):

list\_1.append(10 \*\* ex)

list\_2 = [10 \*\* ex for ex in range(6)]

print(list\_1)

print(list\_2)

There are two parts inside the code, both creating a list containing a few of the first natural powers of ten.

The former uses a routine way of utilizing the for loop, while the latter makes use of the **list comprehension** and builds the list in situ, without needing a loop, or any other extended code.

It looks like the list is created inside itself - it's not true, of course, as Python has to perform nearly the same operations as in the first snippet, but it is indisputable that the second formalism is simply more elegant, and lets the reader avoid any unnecessary details.

The example outputs two identical lines containing the following text:

[1, 10, 100, 1000, 10000, 100000]

[1, 10, 100, 1000, 10000, 100000]

## 4.1.1.7 Generators and closures: More about list comprehensions

There is a very interesting syntax we want to show you now. Its usability is not limited to list comprehensions, but we have to admit that comprehensions are the ideal environment for it. It's a **conditional expression - a way of selecting one of two different values based on the result of a Boolean expression.** Look:

*expression\_one if condition else expression\_two*

It may look a bit surprising at first glance, but you have to keep in mind that it is **not a conditional instruction**. Moreover, it's not an instruction at all. It's an operator.

The value it provides is equal to expression\_one when the condition is True, and expression\_two otherwise. An example will tell more. Look at the code in the editor.

the\_list = []

for x in range(10):

the\_list.append(1 if x % 2 == 0 else 0)

print(the\_list)

The code fills a list with 1's and 0s - if the index of a particular element is odd, the element is set to 0, and to 1 otherwise. Simple? Maybe not at first glance. Elegant? Indisputably. Can you use the same trick within a list comprehension? Yes, you can.

## 4.1.1.8 Generators and closures: More about list comprehensions

Look at the example in the editor.

the\_list = [1 if x % 2 == 0 else 0 for x in range(10)]

print(the\_list)

Compactness and elegance - these two words come to mind when looking at the code.

So, what do they have in common, generators and list comprehensions? Is there any connection between them? Yes. A rather loose connection, but an unequivocal one. Just one change can **turn any list comprehension into a generator.**

**List comprehensions vs. generators**

Now look at the code below and see if you can find the detail that turns a list comprehension into a generator:

the\_list = [1 if x % 2 == 0 else 0 for x in range(10)]

the\_generator = (1 if x % 2 == 0 else 0 for x in range(10))

for v in the\_list:

print(v, end=" ")

print()

for v in the\_generator:

print(v, end=" ")

print()

It's the **parentheses**. The brackets make a comprehension, the parentheses make a generator. The code, however, when run, produces two identical lines:

1 0 1 0 1 0 1 0 1 0

1 0 1 0 1 0 1 0 1 0

How can you know that the second assignment creates a generator, not a list? There is some proof we can show you. Apply the len() function to both these entities.

len(the\_list) will evaluate to 10. Clear and predictable. len(the\_generator) will raise an exception, and you will see the following message:

TypeError: object of type 'generator' has no len()

Of course, saving either the list or the generator is not necessary - you can create them exactly in the place where you need them - just like here:

for v in [1 if x % 2 == 0 else 0 for x in range(10)]:

print(v, end=" ")

print()

for v in (1 if x % 2 == 0 else 0 for x in range(10)):

print(v, end=" ")

print()

Note: the same appearance of the output doesn't mean that both loops work in the same way. In the first loop, the list is created (and iterated through) as a whole - it actually exists when the loop is being executed. In the second loop, there is no list at all - there are only subsequent values produced by the generator, one by one.

## 4.1.1.9 Generators and closures: The lambda function

The lambda function is a concept borrowed from mathematics, more specifically, from a part called the *Lambda calculus*, but these two phenomena are not the same.

Mathematicians use the *Lambda calculus* in many formal systems connected with logic, recursion, or theorem provability. Programmers use the lambda function to simplify the code, to make it clearer and easier to understand.

A lambda function is a function without a name (you can also call it an **anonymous function**). Of course, such a statement immediately raises the question: how do you use anything that cannot be identified?

Fortunately, it's not a problem, as you can name such a function if you really need, but, in fact, in many cases the lambda function can exist and work while remaining fully incognito. The declaration of the lambda function doesn't resemble a normal function declaration in any way - see for yourself:

lambda parameters: expression

Such a clause **returns the value of the expression when taking into account the current value of the current lambda argument.**

As usual, an example will be helpful. Our example uses three lambda functions, but gives them names. Look at it carefully:

two = lambda: 2

sqr = lambda x: x \* x

pwr = lambda x, y: x \*\* y

for a in range(-2, 3):

print(sqr(a), end=" ")

print(pwr(a, two()))

Let's analyze it:

* the first lambda is an anonymous **parameterless function** that always returns 2. As we've **assigned it to a variable named two**, we can say that the function is not anonymous anymore, and we can use the name to invoke it.
* the second one is a **one-parameter anonymous function** that returns the value of its squared argument. We've named it as such, too.
* the third lambda **takes two parameters** and returns the value of the first one raised to the power of the second one. The name of the variable which carries the lambda speaks for itself. We don't use pow to avoid confusion with the built-in function of the same name and the same purpose.

The program produces the following output:

4 4

1 1

0 0

1 1

4 4

This example is clear enough to show how lambdas are declared and how they behave, but it says nothing about why they're necessary, and what they're used for, since they can all be replaced with routine Python functions. Where is the benefit?

## 4.1.1.10 Generators and closures: How to use lambdas and what for?

The most interesting part of using lambdas appears when you can use them in their pure form - **as anonymous parts of code intended to evaluate a result.**

Imagine that we need a function (we'll name it print\_function) which prints the values of a given (other) function for a set of selected arguments. We want print\_function to be universal - it should accept a set of arguments put in a list and a function to be evaluated, both as arguments - we don't want to hardcode anything. Look at the example in the editor. This is how we've implemented the idea.

def print\_function(args, fun):

for x in args:

print('f(', x,')=', fun(x), sep='')

def poly(x):

return 2 \* x\*\*2 - 4 \* x + 2

print\_function([x for x in range(-2, 3)], poly)

Let's analyze it. The print\_function() function takes two parameters:

* the first, a list of arguments for which we want to print the results;
* the second, a function which should be invoked as many times as the number of values that are collected inside the first parameter.

Note: we've also defined a function named poly() - this is the function whose values we're going to print. The calculation the function performs isn't very sophisticated - it's the polynomial (hence its name) of a form:

f(x) = 2x^2 - 4x + 2

The name of the function is then passed to the print\_function() along with a set of five different arguments - the set is built with a list comprehension clause. The code prints the following lines:

f(-2)=18

f(-1)=8

f(0)=2

f(1)=0

f(2)=2

Can we avoid defining the poly() function, as we're not going to use it more than once? Yes, we can - this is the benefit a lambda can bring. Look at the example below. Can you see the difference?

def print\_function(args, fun):

for x in args:

print('f(', x,')=', fun(x), sep='')

print\_function([x for x in range(-2, 3)], lambda x: 2 \* x\*\*2 - 4 \* x + 2)

The print\_function() has remained exactly the same, but there is no poly() function. We don't need it anymore, as the polynomial is now directly inside the print\_function() invocation in the form of a lambda defined in the following way:

lambda x: 2 \* x\*\*2 - 4 \* x + 2

The code has become shorter, clearer, and more legible.

Let us show you another place where lambdas can be useful. We'll start with a description of map(), a built-in Python function. Its name isn't too descriptive, its idea is simple, and the function itself is really usable.

## 4.1.1.11 Generators and closures: Lambdas and the map() function 1/16/22

In the simplest of all possible cases, the map() function:

map(function, list)

takes two arguments:

* a function;
* a list.

The above description is extremely simplified, as:

* the second map() argument may be any entity that can be iterated (e.g., a tuple, or just a generator)
* map() can accept more than two arguments.

**The map() function applies the function passed by its first argument to all its second argument's elements, and returns an iterator delivering all subsequent function results.** You can use the resulting iterator in a loop, or convert it into a list using the list() function. Can you see a role for any lambda here? Look at the code in the editor - we've used two lambdas in it.

list\_1 = [x for x in range(5)]

list\_2 = list(map(lambda x: 2 \*\* x, list\_1))

print(list\_2)

for x in map(lambda x: x \* x, list\_2):

print(x, end=' ')

print()

This is the intrigue:

* build the list\_1 with values from 0 to 4;
* next, use map along with the first lambda to create a new list in which all elements have been evaluated as 2 raised to the power taken from the corresponding element from list\_1;
* list\_2 is printed then;
* in the next step, use the map() function again to make use of the generator it returns and to directly print all the values it delivers; as you can see, we've engaged the second lambda here - it just squares each element from list\_2.

Try to imagine the same code without lambdas. Would it be any better? It's unlikely.

## 4.1.1.12 Generators and closures: Lambdas and the filter() function

Another Python function which can be significantly beautified by the application of a lambda is *filter*(). It expects the same kind of arguments as map(), but does something different - it **filters its second argument while being guided by directions flowing from the function specified as the first argument** (the function is invoked for each list element, just like in map()). The elements which return True from the function **pass the filter** - the others are rejected. The example in the editor shows the filter() function in action.

from random import seed, randint

seed()

data = [randint(-10,10) for x in range(5)]

filtered = list(filter(lambda x: x > 0 and x % 2 == 0, data))

print(data)

print(filtered)

Note: we've made use of the random module to initialize the random number generator (not to be confused with the generators we've just talked about) with the seed() function, and to produce five random integer values from -10 to 10 using the randint() function. The list is then filtered, and only the numbers which are even and greater than zero are accepted.

Of course, it's not likely that you'll receive the same results, but this is what our results looked like:

[6, 3, 3, 2, -7]

[6, 2]

## 4.1.1.13 Generators and closures: A brief look at closures

Let's start with a definition: **closure is a technique which allows the storing of values in spite of the fact that the context in which they have been created does not exist anymore.** Intricate? A bit. Let's analyze a simple example:

def outer(par):

loc = par

var = 1

outer(var)

print(var)

print(loc)

The example is obviously erroneous. The last two lines will cause a NameError exception - neither par nor loc is accessible outside the function. Both the variables exist when and only when the outer() function is being executed. Look at the example in the editor. We've modified the code significantly.

def outer(par):

loc = par

def inner():

return loc

return inner

var = 1

fun = outer(var)

print(fun())

There is a brand new element in it - a function (named inner) inside another function (named outer).

How does it work? Just like any other function except for the fact that inner() may be invoked only from within outer(). We can say that inner() is outer()'s private tool - no other part of the code can access it. Look carefully:

* the inner() function returns the value of the variable accessible inside its scope, as inner() can use any of the entities at the disposal of outer()
* the outer() function returns the inner() function itself; more precisely, it returns a copy of the inner() function, the one which was frozen at the moment of outer()'s invocation; the frozen function contains its full environment, including the state of all local variables, which also means that the value of loc is successfully retained, although outer() ceased to exist a long time ago.

In effect, the code is fully valid, and outputs: 1

The function returned during the outer() invocation is a **closure**.

## 4.1.1.14 Generators and closures: A brief look at closures: continued

**A closure has to be invoked in exactly the same way in which it has been declared.** In the example below:

def outer(par):

loc = par

def inner():

return loc

return inner

var = 1

fun = outer(var)

print(fun())

the inner() function is parameterless, so we have to invoke it without arguments. Now look at the code in the editor. It is fully possible to **declare a closure equipped with an arbitrary number of parameters**, e.g., one, just like the power() function.

def make\_closure(par):

loc = par

def power(p):

return p \*\* loc

return power

fsqr = make\_closure(2)

fcub = make\_closure(3)

for i in range(5):

print(i, fsqr(i), fcub(i))

This means that the closure not only makes use of the frozen environment, but it can also **modify its behavior by using values taken from the outside.**

This example shows one more interesting circumstance - you can **create as many closures as you want using one and the same piece of code.** This is done with a function named make\_closure(). Note:

* the first closure obtained from make\_closure() defines a tool squaring its argument;
* the second one is designed to cube the argument.

This is why the code produces the following output:

0 0 0

1 1 1

2 4 8

3 9 27

4 16 64

Carry out your own tests.

## 4.1.1.15 SECTION SUMMARY: Key takeaways

1. An **iterator** is an object of a class providing at least **two** methods (not counting the constructor!):

* \_\_iter\_\_() is invoked once when the iterator is created and returns the iterator's object **itself**;
* \_\_next\_\_() is invoked to provide the **next iteration's value** and raises the StopIteration exception when the iteration **comes to and end.**

2. The *yield* statement can be used only inside functions. The *yield* statement suspends function execution and causes the function to return the yield's argument as a result. Such a function cannot be invoked in a regular way – its only purpose is to be used as a **generator** (i.e. in a context that requires a series of values, like a for loop.)

3. A **conditional expression** is an expression built using the if-else operator. For example: print(True if 0 >=0 else False) #outputs True.

4. A **list comprehension** becomes a **generator** when used inside **parentheses** (used inside brackets, it produces a regular list). For example:

for x in (el \* 2 for el in range(5)):

print(x) #outputs 0 2 4 6 8.

4. A **lambda function** is a tool for creating anonymous functions. For example:

def foo(x,f):

return f(x)

print(foo(9, lambda x: x \*\* 0.5)) # outputs 3.0.

5. The map(fun, list) function creates a **copy** of a list argument, and applies the fun function to all of its elements, returning a **generator** that provides the new list content element by element. For example:

short\_list = ['mython', 'python', 'fell', 'on', 'the', 'floor']

new\_list = list(map(lambda s: s.title(), short\_list))

print(new\_list)

outputs ['Mython', 'Python', 'Fell', 'On', 'The', 'Floor'].

6. The filter(fun, list) function creates a **copy** of those list elements, which cause the fun function to return True. The function's result is a **generator** providing the new list content element by element. For example:

short\_list = [1, "Python", -1, "Monty"]

new\_list = list(filter(lambda s: isinstance(s, str), short\_list))

print(new\_list) # outputs ['Python', 'Monty'].

7. A closure is a technique which allows the **storing of values** in spite of the fact that the **context** in which they have been created **does not exist anymore**. For example:

def tag(tg):

tg2 = tg

tg2 = tg[0] + '/' + tg[1:]

def inner(str):

return tg + str + tg2

return inner

b\_tag = tag('<b>')

print(b\_tag('Monty Python')) # outputs <b>Monty Python</b>

**Exercise 1: What is the expected output of the following code?**

class Vowels:

def \_\_init\_\_(self):

self.vow = "aeiouy " # Yes, we know y isn’t always considered a vowel.

self.pos = 0

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.pos == len(self.vow):

raise StopIteration

self.pos += 1

return self.vow[self.pos - 1]

vowels = Vowels()

for v in vowels:

print(v, end=' ') # Check: a e i o u y

**Exercise 2: Write a lambda function, setting the least significant bit of its integer argument, and apply it to the map() function to produce the string 1 3 3 5 on the console.**

any\_list = [1, 2, 3, 4]

even\_list = # Complete the line here.

print(even\_list) # Check: list(map(lambda n: n | 1, any\_list))

**Exercise 3: What is the expected output of the following code?**

def replace\_spaces(replacement='\*'):

def new\_replacement(text):

return text.replace(' ', replacement)

return new\_replacement

stars = replace\_spaces()

print(stars("And Now for Something Completely Different"))

Check: And\*Now\*for\*Something\*Completely\*Different

Note: PEP 8, the Style Guide for Python Code, recommends that **lambdas should not be assigned to variables, but rather they should be defined as functions.**

This means that it is better to use a def statement, and avoid using an assignment statement that binds a lambda expression to an identifer. For example:

# Recommended:

def f(x): return 3\*x

# Not recommended:

f = lambda x: 3\*x

Binding lambdas to identifiers generally duplicates the functionality of the def statement. Using def statements, on the other hand, generates more lines of code.

It is important to understand that reality often likes to draw its own scenarios, which do not necessarily follow the conventions or formal recommendations. Whether you decide to follow them or not will depend on many things: your preferences, other conventions adopted, company internal guidelines, compatibility with existing code, etc. Be aware of this.

# **4.2. FILES (File Streams, File Processing, Diagnosing Streaming Problems)**

## 4.2.1.1 Processing files:Accessing files from Python code

One of the most common issues in the developer's job is to **process data stored in files** while the files are usually physically stored using storage devices - hard, optical, network, or solid-state disks. It's easy to imagine a program that sorts 20 numbers, and it's equally easy to imagine the user of this program entering these twenty numbers directly from the keyboard. It's much harder to imagine the same task when there are 20,000 numbers to be sorted, and there isn't a single user who is able to enter these numbers without making a mistake. It's much easier to imagine that these numbers are stored in the disk file which is read by the program. The program sorts the numbers and doesn't send them to the screen, but instead creates a new file and saves the sorted sequence of numbers there.

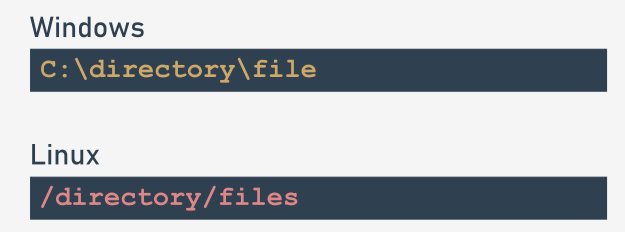
If we want to implement a simple database, the only way to store the information between program runs is to save it into a file (or files if your database is more complex).

In principle, any non-simple programming problem relies on the use of files, whether it processes images (stored in files), multiplies matrices (stored in files), or calculates wages and taxes (reading data stored in files).

You may ask why we have waited until now to show you these issues. The answer is very simple - Python's way of accessing and processing files is implemented using a consistent set of objects. There is no better moment to talk about it.

## 4.2.1.2 Processing files: File names

Different operating systems can treat the files in different ways. For example, Windows uses a different naming convention than the one adopted in Unix/Linux systems. If we use the notion of a canonical file name (a name which uniquely defines the location of the file regardless of its level in the directory tree) we can realize that these names look different in Windows and in Unix/Linux:

As you can see, systems derived from Unix/Linux don't use the disk drive letter (e.g., C:) and all the directories grow from one root directory called /, while Windows systems recognize the root directory as \.

In addition, Unix/Linux system file names are case-sensitive. Windows systems store the case of letters used in the file name, but don't distinguish between their cases at all. This means that these two strings: ThisIsTheNameOfTheFile and thisisthenameofthefile describe two different files in Unix/Linux systems, but are the same name for just one file in Windows systems.

The main and most striking difference is that you have to use **two different separators for the directory names**: \ in Windows, and / in Unix/Linux.

This difference is not very important to the normal user, but is **very important when writing programs in Python.** To understand why, try to recall the very specific role played by the \ inside Python strings.

## 4.2.1.3 Processing files: File names: continued

Suppose you're interested in a particular file located in the directory dir, and named file. Suppose also that you want to assign a string containing the name of the file. In Unix/Linux systems, it may look as follows:

name = "/dir/file"

But if you try to code it for the Windows system:

name = "\dir\file"

you'll get an unpleasant surprise: either Python will generate an error, or the execution of the program will behave strangely, as if the file name has been distorted in some way. In fact, it's not strange at all, but quite obvious and natural. Python uses the \ as an escape character (like \n). This means that Windows file names must be written as follows:

name = "\\dir\\file"

Fortunately, there is also one more solution. Python is smart enough to be able to convert slashes into backslashes each time it discovers that it's required by the OS. This means that any the following assignments:

name = "/dir/file"

name = "c:/dir/file"

will work with Windows, too. Any program written in Python (and not only in Python, because that convention applies to virtually all programming languages) does not communicate with the files directly, but through some abstract entities that are named differently in different languages or environments - the most-used terms are **handles** or **streams** (we'll use them as synonyms here).

The programmer, having a more- or less-rich set of functions/methods, is able to perform certain operations on the stream, which affect the real files using mechanisms contained in the operating system kernel. In this way, you can implement the process of accessing any file, even when the name of the file is unknown at the time of writing the program.

The operations performed with the abstract stream reflect the activities related to the physical file.

To connect (bind) the stream with the file, it's necessary to perform an explicit operation. The operation of connecting the stream with a file is called **opening the file**, while disconnecting this link is named **closing the file**.

Hence, the conclusion is that the very first operation performed on the stream is always *open* and the last one is *close*. The program, in effect, is free to manipulate the stream between these two events and to handle the associated file.

This freedom is limited, of course, by the physical characteristics of the file and the way in which the file has been opened.

Let us say again that the opening of the stream can fail, and it may happen due to several reasons: the most common is the lack of a file with a specified name.

It can also happen that the physical file exists, but the program is not allowed to open it. There's also the risk that the program has opened too many streams, and the specific operating system may not allow the simultaneous opening of more than n files (e.g., 200). A well-written program should detect these failed openings, and react accordingly.

## 4.2.1.4 Processing files: File streams

The opening of the stream is not only associated with the file, but should also declare the manner in which the stream will be processed. This declaration is called an **open mode.** If the opening is successful, **the program will be allowed to perform only the operations which are consistent with the declared open mode.**

There are two basic operations performed on the stream:

* **read** from the stream: the portions of the data are retrieved from the file and placed in a memory area managed by the program (e.g., a variable);
* **write** to the stream: the portions of the data from the memory (e.g., a variable) are transferred to the file.

There are three basic modes used to open the stream:

* **read mode**: a stream opened in this mode allows **read operations only**; trying to write to the stream will cause an exception (the exception is named UnsupportedOperation, which inherits OSError and ValueError, and comes from the io module);
* **write mode**: a stream opened in this mode allows **write operations only**; attempting to read the stream will cause the exception mentioned above;
* **update mode**: a stream opened in this mode allows both **writes and reads**.

Before we discuss how to manipulate the streams, we owe you some explanation. **The stream behaves almost like a tape recorder.** When you read something from a stream, a virtual head moves over the stream according to the number of bytes transferred from the stream.When you write something to the stream, the same head moves along the stream recording the data from the memory.

Whenever we talk about reading from and writing to the stream, try to imagine this analogy. The programming books refer to this mechanism as the **current file position**, and we'll also use this term.

Before we discuss how to manipulate the streams, we owe you some explanation. The stream behaves almost like a tape recorder.

When you read something from a stream, a virtual head moves over the stream according to the number of bytes transferred from the stream.

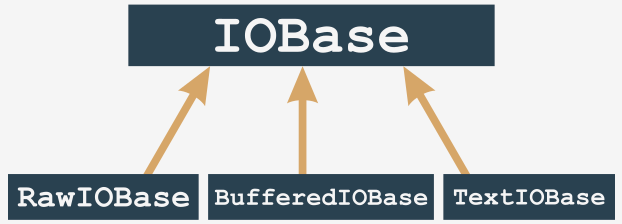
When you write something to the stream, the same head moves along the stream recording the data from the memory.

Whenever we talk about reading from and writing to the stream, try to imagine this analogy. The programming books refer to this mechanism as the current file position, and we'll also use this term.

## 4.2.1.5 Processing files: File handles

Python assumes that **every file is hidden behind an object of an adequate class.** Of course, it's hard not to ask how to interpret the word *adequate*. Files can be processed in many different ways - some of them depend on the file's contents, some on the programmer's intentions. In any case, different files may require different sets of operations, and behave in different ways.

An object of an adequate class is **created when you open the file and annihilate(消灭) it at the time of closing.** Between these two events, you can use the object to specify what operations should be performed on a particular stream. The operations you're allowed to use are imposed by **the way in which you've opened the file.** In general, the object comes from one of the classes shown here:

Note: you never use constructors to bring these objects to life. The only way you **obtain them is to invoke the function named open().** The function analyses the arguments you've provided, and automatically creates the required object.

If you want to **get rid of the object, you invoke the method named close().** The invocation will sever the connection to the object, and the file and will remove the object. For our purposes, we'll concern ourselves only with streams represented by BufferIOBase and TextIOBase objects. You'll understand why soon.

## 4.2.1.6 Processing files: File handles: continued

Due to the type of the stream's contents, **all the streams are divided into text and binary streams.** The text streams ones are structured in lines; that is, they contain typographical characters (letters, digits, punctuation, etc.) arranged in rows (lines), as seen with the naked eye when you look at the contents of the file in the editor. This file is written (or read) mostly character by character, or line by line. The binary streams don't contain text but a sequence of bytes of any value. This sequence can be, for example, an executable program, an image, an audio or a video clip, a database file, etc.

Because these files don't contain lines, the reads and writes relate to portions of data of any size. Hence the data is read/written byte by byte, or block by block, where the size of the block usually ranges from one to an arbitrarily chosen value.

Then comes a subtle problem. In Unix/Linux systems, the line ends are marked by a single character named LF (ASCII code 10) designated in Python programs as \n.

Other operating systems, especially these derived from the prehistoric CP/M system (which applies to Windows family systems, too) use a different convention: the end of line is marked by a pair of characters, CR and LF (ASCII codes 13 and 10) which can be encoded as \r\n. This ambiguity can cause various unpleasant consequences.

If you create a program responsible for processing a text file, and it is written for Windows, you can recognize the ends of the lines by finding the \r\n characters, but the same program running in a Unix/Linux environment will be completely useless, and vice versa: the program written for Unix/Linux systems might be useless in Windows.

Such undesirable features of the program, which prevent or hinder the use of the program in different environments, are called **non-portability**. Similarly, the trait of the program allowing execution in different environments is called **portability**. A program endowed with such a trait is called a **portable program**.

## 4.2.1.7 Processing files: File handles: continued

Since portability issues were (and still are) very serious, a decision was made to definitely resolve the issue in a way that doesn't engage the developer's attention.

It was done at the level of classes, which are responsible for reading and writing characters to and from the stream. It works in the following way:

* when the stream is open and it's advised that the data in the associated file will be processed as text (or there is no such advisory at all), it is **switched into text mode**;
* during reading/writing of lines from/to the associated file, nothing special occurs in the Unix environment, but when the same operations are performed in the Windows environment, a process called a **translation of newline characters** occurs: when you read a line from the file, every pair of \r\n characters is replaced with a single \n character, and vice versa; during write operations, every \n character is replaced with a pair of \r\n characters;
* the mechanism is completely **transparent** to the program, which can be written as if it was intended for processing Unix/Linux text files only; the source code run in a Windows environment will work properly, too;
* when the stream is open and it's advised to do so, its contents are taken as-is, **without any conversion** - no bytes are added or omitted.

**Opening the streams**

The opening of the stream is performed by a function which can be invoked in the following way:

stream = open(file, mode = 'r', encoding = None)

Let's analyze it:

* the name of the function (open) speaks for itself; if the opening is successful, the function returns a stream object; otherwise, an exception is raised (e.g., FileNotFoundError **if the file you're going to read doesn't exist)**;
* the first parameter of the function (file) specifies the name of the file to be associated with the stream;
* the second parameter (mode) specifies the open mode used for the stream; it's a string filled with a sequence of characters, and each of them has its own special meaning (more details soon);
* the third parameter (encoding) specifies the encoding type (e.g., UTF-8 when working with text files)
* the opening must be the very first operation performed on the stream.

Note: the mode and encoding arguments may be omitted - their default values are assumed then. The default opening mode is reading in text mode, while the default encoding depends on the platform used. Let’s present the most important and useful open modes. Ready?

## 4.2.1.8 Processing files: Opening the streams: modes

r open mode: read

* the stream will be opened in **read mode**;
* the file associated with the stream **must exist** and has to be readable, otherwise the open() function raises an exception.

w open mode: write

* the stream will be opened in **write mode**;
* the file associated with the stream **doesn't need to exist**; if it doesn't exist it will be created; if it exists, it will be truncated to the length of zero (erased); if the creation isn't possible (e.g., due to system permissions) the open() function raises an exception.

a open mode: append

* the stream will be opened in **append mode**;
* the file associated with the stream **doesn't need to exist**; if it doesn't exist, it will be created; if it exists the virtual recording head will be set at the end of the file (the previous content of the file remains untouched.)

r+ open mode: read and update

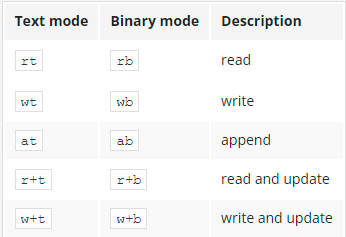
* the stream will be opened in **read and update mode**;
* the file associated with the stream **must exist and has to be writeable**, otherwise the open() function raises an exception;
* both read and write operations are allowed for the stream.

w+ open mode: write and update

* the stream will be opened in **write and update** mode;
* the file associated with the stream **doesn't need to exist**; if it doesn't exist, it will be created; the previous content of the file remains untouched;
* both read and write operations are allowed for the stream.

**Selecting text and binary modes**

If there is a letter b at the end of the mode string it means that the stream is to be opened in the **binary mode**. If the mode string ends with a letter t the stream is opened in the **text mode**. Text mode is the default behaviour assumed when no binary/text mode specifier is used. Finally, the successful opening of the file will set the current file position (the virtual reading/writing head) before the first byte of the file **if the mode is not** a and after the last byte of file **if the mode is set to** a.



Extra: You can also open a file for its exclusive creation. You can do this using the x open mode. If the file already exists, the open() function will raise an exception.

## 4.2.1.9 Processing files: Opening the stream for the first time

Imagine that we want to develop a program that reads content of the text file named: C:\Users\User\Desktop\file.txt. How to open that file for reading? Here's the relevant snippet of the code:

try:

stream = open("C:\Users\User\Desktop\file.txt", "rt")

# Processing goes here.

stream.close()

except Exception as exc:

print("Cannot open the file:", exc)

What's going on here?

* we open the try-except block as we want to handle runtime errors softly;
* we use the open() function to try to open the specified file (note the way we've specified the file name)
* the open mode is defined as text to read (as **text is the default setting**, we can skip the t in mode string)
* in case of success we get an object from the open() function and we assign it to the stream variable;
* if open() fails, we handle the exception printing full error information (it's definitely good to know what exactly happened)

**Pre-opened streams**

We said earlier that any stream operation must be preceded by the open() function invocation. There are three well-defined exceptions to the rule.

When our program starts, the three streams are already opened and don't require any extra preparations. What's more, your program can use these streams explicitly if you take care to import the sys module:

import sys

because that's where the declaration of the three streams is placed.

The names of these streams are: sys.stdin, sys.stdout, and sys.stderr. Let's analyze them:

* sys.stdin
* stdin (as standard input)
* the stdin stream is normally associated with the keyboard, pre-open for reading and regarded as the primary data source for the running programs;
* the well-known input() function reads data from stdin by default.
* sys.stdout
* stdout (as standard output)
* the stdout stream is normally associated with the screen, pre-open for writing, regarded as the primary target for outputting data by the running program;
* the well-known print() function outputs the data to the stdout stream.
* sys.stderr
* stderr (as standard error output)
* the stderr stream is normally associated with the screen, pre-open for writing, regarded as the primary place where the running program should send information on the errors encountered during its work;
* we haven't presented any method to send the data to this stream (we will do it soon, we promise)
* the separation of stdout (useful results produced by the program) from the stderr (error messages, undeniably useful but does not provide results) gives the possibility of redirecting these two types of information to the different targets. More extensive discussion of this issue is beyond the scope of our course. The operation system handbook will provide more information on these issues.

## 4.2.1.10 Processing files: Closing streams 1/17/22

The last operation performed on a stream (this doesn't include the stdin, stdout, and stderr streams which don't require it) should be **closing**. That action is performed by a method invoked from within open stream object: stream.close().

* the name of the function is definitely self-commenting: close()
* the function expects exactly no arguments; the stream doesn't need to be opened
* the function returns nothing but raises IOError exception in case of error;
* most developers believe that the close() function always succeeds and thus there is no need to check if it's done its task properly.

This belief is only partly justified. If the stream was opened for writing and then a series of write operations were performed, it may happen that the data sent to the stream has not been transferred to the physical device yet (due to mechanism called **caching** or **buffering**).

Since the closing of the stream forces the buffers to flush them, it may be that the flushes fail and therefore the close() fails too.We have already mentioned failures caused by functions operating with streams but not mentioned a word how exactly we can identify the cause of the failure. The possibility of making a diagnosis exists and is provided by one of streams' exception component which we are going to tell you about just now.

**Diagnosing stream problems**

The IOError object is equipped with a property named errno (the name comes from the phrase error number) and you can access it as follows:

try:

# Some stream operations.

except IOError as exc:

print(exc.errno)

The value of the errno attribute can be compared with one of the predefined symbolic constants defined in the errno module. Let's take a look at some selected constants useful for detecting stream errors:

* errno.EACCES → Permission denied (The error occurs when you try, for example, to open a file with the read only attribute for writing.)
* errno.EBADF → Bad file number(The error occurs when you try, for example, to operate with an unopened stream.)
* errno.EEXIST → File exists(The error occurs when you try, for example, to rename a file with its previous name.)
* errno.EFBIG → File too large(The error occurs when you try to create a file that is larger than the maximum allowed by the operating system.)
* errno.EISDIR → Is a directory(The error occurs when you try to treat a directory name as the name of an ordinary file.)
* errno.EMFILE → Too many open files(The error occurs when you try to simultaneously open more streams than acceptable for your operating system.)
* errno.ENOENT → No such file or directory(The error occurs when you try to access a non-existent file/directory.)
* errno.ENOSPC → No space left on device(The error occurs when there is no free space on the media.)

The complete list is much longer (it includes also some error codes not related to the stream processing.)

## 4.2.1.11 Processing files: Diagnosing stream problems: continued

If you are a very careful programmer, you may feel the need to use the sequence of statements similar to those presented in the editor.

import errno

try:

s = open("c:/users/user/Desktop/file.txt", "rt")

# Actual processing goes here.

s.close()

except Exception as exc:

if exc.errno == errno.ENOENT:

print("The file doesn't exist.")

elif exc.errno == errno.EMFILE:

print("You've opened too many files.")

else:

print("The error number is:", exc.errno)

Fortunately, there is a function that can dramatically **simplify the error handling code.**

Its name is strerror(), and it comes from the os module and **expects just one argument - an error number.** Its role is simple: you give an error number and get a string describing the meaning of the error.

Note: if you pass a non-existent error code (a number which is not bound to any actual error), the function will raise ValueError exception.

Now we can simplify our code in the following way:

from os import strerror

try:

s = open("c:/users/user/Desktop/file.txt", "rt")

# Actual processing goes here.

s.close()

except Exception as exc:

print("The file could not be opened:", strerror(exc.errno))

Okay. Now it's time to deal with text files and get familiar with some basic techniques you can use to process them.

## 4.2.1.12 Section Summary: Key takeaways

1. A file needs to be **open** before it can be processed by a program, and it should be **closed** when the processing is finished.

Opening the file associates it with the **stream**, which is an abstract representation of the physical data stored on the media. The way in which the stream is processed is called **open mode**. Three open modes exist:

* **read mode** – only read operations are allowed;
* **write mode** – only write operations are allowed;
* **update mode** – both writes and reads are allowed.

2. Depending on the physical file content, different Python classes can be used to process files. In general, the BufferedIOBase is able to process any file, while TextIOBase is a specialized class dedicated to processing text files (i.e. files containing human-visible texts divided into lines using new-line markers). Thus, the streams can be divided into **binary** and **text** ones.

3. The following open() function syntax is used to open a file:

open(file\_name, mode=open\_mode, encoding=text\_encoding)

The invocation creates a stream object and associates it with the file named file\_name, using the specified open\_mode and setting the specified text\_encoding, or it **raises an exception in the case of an error.**

4. Three predefined streams are already open when the program starts:

* sys.stdin – standard input;
* sys.stdout – standard output;
* sys.stderr – standard error output.

4. The IOError exception object, created when any file operations fails (including open operations), contains a property named errno, which contains the completion code of the failed action. Use this value to diagnose the problem.

Exercise 1: How do you encode an open() function’s mode argument value if you're going to create a new text file to only fill it with an article?

Check: "wt" or "w"

**Exercise 2: What is the meaning of the value represented by errno.EACCESS?**

Check: Permission denied: you're not allowed to access the file's content.

**Exercise 3:What is the expected output of the following code, assuming that the file named file does not exist?**

import errno

try:

stream = open("file", "rb")

print("exists")

stream.close()

except IOError as error:

if error.errno == errno.ENOENT:

print("absent")

else:

print("unknown")

Check: absent

# **4.3. PROCESSING TEXT AND BINARY FILES**

## 4.3.1.1 Working with real files: Processing text files

In this lesson we're going to prepare a simple text file with some short, simple content. We're going to show you some basic techniques you can utilize to **read the file contents** in order to process them. The processing will be very simple - you're going to copy the file's contents to the console, and count all the characters the program has read in.

But remember - our understanding of a text file is very strict. In our sense, it's a plain text file - it may contain only text, without any additional decorations (formatting, different fonts, etc.). That's why you should avoid creating the file using any advanced text processor like MS Word, LibreOffice Writer, or something like this. Use the very basics your OS offers: Notepad, vim, gedit, etc. If your text files contain some national characters not covered by the standard ASCII charset, you may need an additional step. Your open() function invocation may require an argument denoting specific text encoding.

For example, if you're using a Unix/Linux OS configured to use UTF-8 as a system-wide setting, the open() function may look as follows:

stream = open('file.txt', 'rt', encoding='utf-8')

where the encoding argument has to be set to a value which is a string representing proper text encoding (UTF-8, here). Consult your OS documentation to find an encoding name adequate to your environment.

Note: For the purposes of our experiments with file processing carried out in this section, we're going to use a pre-uploaded set of files (i.e., tzop.txt, or text.txt files) which you'll be able to work with. If you'd like to work with your own files locally on your machine, we strongly encourage you to do so, and to use IDLE (or any other IDE that you may prefer) to carry out your own tests.

# Opening tzop.txt in read mode, returning it as a file object:

stream = open("tzop.txt", "rt", encoding = "utf-8")

print(stream.read()) # printing the content of the file

Output:

The Zen of Python

=================

::

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

:

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

----------------

source: https://github.com/python/peps/blob/master/pep-0020.txt

## 4.3.1.2 Working with real files: Processing text files: continued 1/18/22

Reading a text file's contents can be performed using several different methods - none of them is any better or worse than any other. It's up to you which of them you prefer and like. Some of them will sometimes be handier, and sometimes more troublesome. Be flexible. Don't be afraid to change your preferences.

The most basic of these methods is the one offered by the *read*() function, which you were able to see in action in the previous lesson. If applied to a text file, the function can:

* read a desired number of characters (including just one) from the file, and return them as a string;
* read all the file contents, and return them as a string;
* if there is nothing more to read (the virtual reading head reaches the end of the file), the function returns an empty string.

We'll start with the simplest variant and use a file named text.txt. The file has the following contents:

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

text.txt

Now look at the code in the editor, and let's analyze it.

from os import strerror

try:

cnt = 0

s = open('text.txt', "rt")

ch = s.read(1)

while ch != '':

print(ch, end='')

cnt += 1

ch = s.read(1)

s.close()

print("\n\nCharacters in file:", cnt)

except IOError as e:

print("I/O error occurred: ", strerror(e.errno))

Output：

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Characters in file: 131

The routine is rather simple:

* use the try-except mechanism and open the file of the predetermined name (text.txt in our case)
* try to read the very first character from the file (ch = s.read(1))
* if you succeed (this is proven by a positive result of the while condition check), output the character (note the end= argument - it's important! You don't want to skip to a new line after every character!);
* update the counter (cnt), too;
* try to read the next character, and the process repeats.

## 4.3.1.3 Working with real files: Processing text files: continued

If you're absolutely sure that the file's length is safe and you can read the whole file to the memory at once, you can do it - the read() function, invoked without any arguments or with an argument that evaluates to None, will do the job for you. **Remember - reading a terabyte-long file using this method may corrupt your OS.** Don't expect miracles - computer memory isn't stretchable. Look at the code in the editor. What do you think of it?

from os import strerror

try:

cnt = 0

s = open('text.txt', "rt")

content = s.read()

for ch in content:

print(ch, end='')

cnt += 1

s.close()

print("\n\nCharacters in file:", cnt)

except IOError as e:

print("I/O error occurred: ", strerr(e.errno))

Let's analyze it:

* open the file as previously;
* read its contents by one read() function invocation;
* next, process the text, iterating through it with a regular for loop, and updating the counter value at each turn of the loop;

The result will be exactly the same as previously.

## 4.3.1.4 Working with real files: Processing text files: readline()

If you want to treat the file's contents **as a set of lines**, not a bunch of characters, the readline() method will help you with that. The method tries to **read a complete line of text from the file**, and returns it as a string in the case of success. Otherwise, it returns an empty string.

This opens up new opportunities - now you can also count lines easily, not only characters. Let's make use of it. Look at the code in the editor.

from os import strerror

try:

ccnt = lcnt = 0

s = open('text.txt', 'rt')

line = s.readline()

while line != '':

lcnt += 1

for ch in line:

print(ch, end='')

ccnt += 1

line = s.readline()

s.close()

print("\n\nCharacters in file:", ccnt)

print("Lines in file: ", lcnt)

except IOError as e:

print("I/O error occurred:", strerror(e.errno))

As you can see, the general idea is exactly the same as in both previous examples.

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Characters in file: 131

Lines in file: 4

## 4.3.1.5 Working with real files: Processing text files: readline()

Another method, which treats text file as a set of lines, not characters, is readlines(). The readlines() method, when invoked without arguments, **tries to read all the file contents, and returns a list of strings, one element per file line.**

If you're not sure if the file size is small enough and don't want to test the OS, you can convince the readlines() method to read not more than a specified number of bytes at once (the returning value remains the same - it's a list of a string). Feel free to experiment with the following example code to understand how the readlines() method works:

s = open("text.txt")

print(s.readlines(20))

print(s.readlines(20))

print(s.readlines(20))

print(s.readlines(20))

s.close()

**The maximum accepted input buffer size is passed to the method as its argument.**

You may expect that readlines() can process a file's contents more effectively than readline(), as it may need to be invoked fewer times.

Note: when there is nothing to read from the file, the method returns an empty list. Use it to detect the end of the file.

To the extent of the buffer's size, you can expect that increasing it may improve input performance, but there is no golden rule for it - try to find the optimal values yourself.

Look at the code in the editor. We've modified it to show you how to use readlines().

from os import strerror

try:

ccnt = lcnt = 0

s = open('text.txt', 'rt')

lines = s.readlines(20)

while len(lines) != 0:

for line in lines:

lcnt += 1

for ch in line:

print(ch, end='')

ccnt += 1

lines = s.readlines(10)

s.close()

print("\n\nCharacters in file:", ccnt)

print("Lines in file: ", lcnt)

except IOError as e:

print("I/O error occurred:", strerror(e.errno))

We've decided to use a 15-byte-long buffer. Don't think it's a recommendation. We've used such a value to avoid the situation in which the first readlines() invocation consumes the whole file. We want the method to be forced to work harder, and to demonstrate its capabilities. There are **two nested loops in the code**: the outer one uses readlines()'s result to iterate through it, while the inner one prints the lines character by character.

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Characters in file: 131

Lines in file: 4

## 4.3.1.6 Working with real files: Processing text files: continued 1/19/22

The last example we want to present shows a very interesting trait of the object returned by the open() function in text mode. We think it may surprise you - **the object is an instance of the iterable class.** Strange? Not at all. Usable? Yes, absolutely.

**The iteration protocol defined for the file object** is very simple - its \_\_next\_\_ method just **returns the next line read in from the file.** Moreover, you can expect that the object automatically invokes close() when any of the file reads reaches the end of the file.Look at the editor and see how simple and clear the code has now become.

from os import strerror

try:

ccnt = lcnt = 0

for line in open('text.txt', 'rt'):

lcnt += 1

for ch in line:

print(ch, end='')

ccnt += 1

print("\n\nCharacters in file:", ccnt)

print("Lines in file: ", lcnt)

except IOError as e:

print("I/O error occurred: ", strerror(e.errno))

Output:

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Characters in file: 131

Lines in file: 4

## 4.3.1.7 Working with real files: Dealing with text files: write()

Writing text files seems to be simpler, as in fact there is one method that can be used to perform such a task. The method is named write() and it expects just one argument - a string that will be transferred to an open file (don't forget - the open mode should reflect the way in which the data is transferred - **writing a file opened in read mode won't succeed**).

No newline character is added to the write()'s argument, so you have to add it yourself if you want the file to be filled with a number of lines.

The example in the editor shows a very simple code that creates a file named newtext.txt (note: the open mode w ensures that the **file will be created from scratch**, even if it exists and contains data) and then puts ten lines into it.

from os import strerror

try:

fo = open('newtext.txt', 'wt') # A new file (newtext.txt) is created.

for i in range(10):

s = "line #" + str(i+1) + "\n"

for ch in s:

fo.write(ch)

fo.close()

except IOError as e:

print("I/O error occurred: ", strerror(e.errno))

The string to be recorded consists of the word line, followed by the line number. We've decided to write the string's contents character by character (this is done by the inner for loop) but you're not obliged to do it in this way. We just wanted to show you that write() is able to operate on single characters. The code creates a file filled with the following text:

line #1

line #2

:

line #9

line #10

Can you print the file's contents to the console? We encourage you to test the behavior of the write() method locally on your machine.

## 4.3.1.8 Working with real files: Dealing with text files: continued

Look at the example in the editor. We've modified the previous code to write whole lines to the text file.

from os import strerror

try:

fo = open('newtext.txt', 'wt')

for i in range(10):

fo.write("line #" + str(i+1) + "\n")

fo.close()

except IOError as e:

print("I/O error occurred: ", strerror(e.errno))

The contents of the newly created file are the same. Note: you can use the same method to write to the stderr stream, but don't try to open it, as it's always open implicitly. For example, if you want to send a message string to stderr to distinguish it from normal program output, it may look like this:

import sys

sys.stderr.write("Error message")

## 4.3.1.9 Working with real files: What is a bytearray?

Before we start talking about binary files, we have to tell you about one of the **specialized classes Python uses to store amorphous(不规则的) data. Amorphous data is data which have no specific shape or form** - they are just a series of bytes.

This doesn't mean that these bytes cannot have their own meaning, or cannot represent any useful object, e.g., bitmap graphics. The most important aspect of this is that in the place where we have contact with the data, we are not able to, or simply don't want to, know anything about it. Amorphous data cannot be stored using any of the previously presented means - they are neither strings nor lists. There should be a special container able to handle such data.

Python has more than one such container - one of them is **a specialized class name bytearray** - as the name suggests, it's **an array containing (amorphous) bytes.**

If you want to have such a container, e.g., in order to read in a bitmap image and process it in any way, you need to create it explicitly, using one of available constructors. Take a look:

data = bytearray(10)

Output: bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')

Such an invocation creates a bytearray object able to store ten bytes.

Note: such a constructor fills the **whole array with zeros.**

## 4.3.1.10 Working with real files: Bytearrays: continued

Bytearrays resemble lists in many respects. For example, they are **mutable**, they're a subject of the len() function, and you can access any of their elements using conventional indexing. There is one important limitation - **you mustn't set any byte array elements with a value which is not an integer** (violating this rule will cause a TypeError exception) and you're **not allowed to assign a value that doesn't come from the range 0 to 255 inclusive** (unless you want to provoke a ValueError exception). You can **treat any byte array elements as integer values** - just like in the example in the editor.

data = bytearray(10)

for i in range(len(data)):

data[i] = 10 - i

for b in data:

print(hex(b))

Output:

0xa

0x9

0x8

0x7

0x6

0x5

0x4

0x3

0x2

0x1

Note: we've used two methods to iterate the byte arrays, and made use of the hex() function to see the elements printed as hexadecimal values.

Now we're going to show you how to **write a byte array to a binary file** - binary, as we don't want to save its readable representation - we want to write a one-to-one copy of the physical memory content, byte by byte.

## 4.3.1.11 Working with real files: Bytearrays: continued

So, how do we write a byte array to a binary file? Look at the code in the editor.

Let's analyze it:

* first, we initialize bytearray with subsequent values starting from 10; if you want the file's contents to be clearly readable, replace 10 with something like ord('a') - this will produce bytes containing values corresponding to the alphabetical part of the ASCII code (don't think it will make the file a text file - it's still binary, as it was created with a wb flag);
* then, we create the file using the open() function - the only difference compared to the previous variants is the open mode containing the b flag;
* the write() method takes its argument (bytearray) and sends it (as a whole) to the file;
* the stream is then closed in a routine way.

The write() method returns a number of successfully written bytes. If the values differ from the length of the method's arguments, it may announce some write errors. In this case, we haven't made use of the result - this may not be appropriate in every case. Try to run the code and analyze the contents of the newly created output file. You're going to use it in the next step.

**How to read bytes from a stream**

Reading from a binary file requires use of a specialized method name *readinto*(), as the method doesn't create a new byte array object, but fills a previously created one with the values taken from the binary file. Note:

* the method returns the number of successfully read bytes;
* the method tries to fill the whole space available inside its argument; if there are more data in the file than space in the argument, the read operation will stop before the end of the file; otherwise, the method's result may indicate that the byte array has only been filled fragmentarily (the result will show you that, too, and the part of the array not being used by the newly read contents remains untouched)

Look at the complete code below:

from os import strerror

data = bytearray(10)

try:

bf = open('file.bin', 'rb')

bf.readinto(data)

bf.close()

for b in data:

print(hex(b), end=' ')

except IOError as e:

print("I/O error occurred:", strerror(e.errno))

Let's analyze it:

* first, we open the file (the one you created using the previous code) with the mode described as rb;
* then, we read its contents into the byte array named data, of size ten bytes;
* finally, we print the byte array contents - are they the same as you expected?

Run the code and check if it's working.

## 4.3.1.12 Working with real files: How to read bytes from a stream

An alternative way of reading the contents of a binary file is offered by the method named read(). Invoked without arguments, it tries to **read all the contents of the file into the memory,** making them a part of a newly created object of the bytes class.

This class has some similarities to bytearray, with the exception of one significant difference - it's **immutable**. Fortunately, there are no obstacles to creating a byte array by taking its initial value directly from the bytes object, just like here:

from os import strerror

try:

bf = open('file.bin', 'rb')

data = bytearray(bf.read())

bf.close()

for b in data:

print(hex(b), end=' ')

except IOError as e:

print("I/O error occurred:", strerror(e.errno))

Be careful - **don't use this kind of read if you're not sure that the file's contents will fit the available memory.**

## 4.3.1.13 Working with real files: How to read bytes from a stream: continued

If the read() method is invoked with an argument, it **specifies the maximum number of bytes to be read.** The method tries to read the desired number of bytes from the file, and the length of the returned object can be used to determine the number of bytes actually read.

You can use the method just like here:

try:

bf = open('file.bin', 'rb')

data = bytearray(bf.read(5))

bf.close()

for b in data:

print(hex(b), end=' ')

except IOError as e:

print("I/O error occurred:", strerror(e.errno))

Note: the first five bytes of the file have been read by the code - the next five are still waiting to be processed.

## 4.3.1.14 Working with real files: Copying files - a simple and functional tool

Now you're going to amalgamate all this new knowledge, add some fresh elements to it, and use it to write a real code which is able to actually copy a file's contents. Of course, the purpose is not to make a better replacement for commands like copy (MS Windows) or cp (Unix/Linux) but to see one possible way of creating a working tool, even if nobody wants to use it. Look at the code in the editor.

**from** os **import** strerror

srcname **=** **input(**"Enter the source file name: "**)**

**try:**

src **=** **open(**srcname**,** 'rb'**)**

**except** **IOError** **as** e**:**

**print(**"Cannot open the source file: "**,** strerror**(**e**.**errno**))**

**exit(**e**.**errno**)**

dstname **=** **input(**"Enter the destination file name: "**)**

**try:**

dst **=** **open(**dstname**,** 'wb'**)**

**except** **Exception** **as** e**:**

**print(**"Cannot create the destination file: "**,** strerror**(**e**.**errno**))**

src**.**close**()**

**exit(**e**.**errno**)**

buffer **=** **bytearray(**65536**)**

total **=** 0

**try:**

readin **=** src**.**readinto**(**buffer**)**

**while** readin **>** 0**:**

written **=** dst**.**write**(**buffer**[:**readin**])**

total **+=** written

readin **=** src**.**readinto**(**buffer**)**

**except** **IOError** **as** e**:**

**print(**"Cannot create the destination file: "**,** strerror**(**e**.**errno**))**

**exit(**e**.**errno**)**

**print(**total**,**'byte(s) succesfully written'**)**

src**.**close**()**

dst**.**close**()**

Let's analyze it:

* lines 3 through 8: ask the user for the name of the file to copy, and try to open it to read; terminate the program execution if the open fails; note: use the exit() function to stop program execution and to pass the completion code to the OS; any completion code other than 0 says that the program has encountered some problems; use the errno value to specify the nature of the issue;
* lines 10 through 16: repeat nearly the same action, but this time for the output file;
* line 18: prepare a piece of memory for transferring data from the source file to the target one; such a transfer area is often called a buffer, hence the name of the variable; the size of the buffer is arbitrary - in this case, we decided to use 64 kilobytes; technically, a larger buffer is faster at copying items, as a larger buffer means fewer I/O operations; actually, there is always a limit, the crossing of which renders no further improvements; test it yourself if you want.
* line 19: count the bytes copied - this is the counter and its initial value;
* line 21: try to fill the buffer for the very first time;
* line 22: as long as you get a non-zero number of bytes, repeat the same actions;
* line 23: write the buffer's contents to the output file (note: we've used a slice to limit the number of bytes being written, as write() always prefer to write the whole buffer)
* line 24: update the counter;
* line 25: read the next file chunk;
* lines 30 through 32: some final cleaning - the job is done.

## 4.3.1.15 LAB: Character frequency histogram

**LAB:**Estimated time: 30-60 minutes Level of difficulty: Medium

**Objectives**

* improving the student's skills in operating with files (reading)
* using data collections for counting numerous data.

**Scenario**

A text file contains some text (nothing unusual) but we need to know how often (or how rare) each letter appears in the text. Such an analysis may be useful in cryptography, so we want to be able to do that in reference to the Latin alphabet. Your task is to write a program which:

* asks the user for the input file's name;
* reads the file (if possible) and counts all the Latin letters (lower- and upper-case letters are treated as equal)
* prints a simple histogram in alphabetical order (only non-zero counts should be presented)

Create a test file for the code, and check if your histogram contains valid results.

Assuming that the test file contains just one line filled with:

aBc

samplefile.txt

the expected output should look as follows:

a -> 1

b -> 1

c -> 1

**Tip**: We think that a dictionary is a perfect data collection medium for storing the counts. The letters may be keys while the counters can be values.

## 4.3.1.16 LAB: Sorted character frequency histogram

## 4.3.1.17 LAB: Evaluating students' results

## 4.3.1.18 SECTION SUMMARY: Key takeaways 1/20/22

1. To read a file’s contents, the following stream methods can be used:

* read(number): reads the number characters/bytes from the file and returns them as a string; is able to read the whole file at once;
* readline() : reads a single line from the text file;
* readlines(number) : reads the number lines from the text file; can read all lines at once;
* readinto(bytearray): reads the bytes from the file and fills the bytearray with them;

2. To write new content into a file, the following stream methods can be used:

* write(string) : writes a string to a text file;
* write(bytearray): writes all the bytes of bytearray to a file;

3. The open() method returns an iterable object which can be used to iterate through all the file's lines inside a for loop. For example:

for line in open("file", "rt"):

print(line, end='')

The code copies the file's contents to the console, line by line. **Note**: the stream closes itself **automatically** when it reaches the end of the file.

**Exercise 1**: What do we expect from the readlines() method when the stream is associated with an empty file?

Check: An empty list (a zero-length list).

**Exercise 2**: What is the following code intended to do?

for line in open("file", "rt"):

for char in line:

if char.lower() not in "aeiouy ":

print(char, end='')

Check: It copies the file's contents to the console, ignoring all vowels.

**Exercise 3**: You're going to process a bitmap stored in a file named image.png, and you want to read its contents as a whole into a bytearray variable named image. Add a line to the following code to achieve this goal.

try:

stream = open("image.png", "rb")

# Insert a line here.

stream.close()

except IOError:

print("failed")

else:

print("success")

Check: image = bytearray(stream.read())

# **4.4. THE OS MODULE - INTERACTING WITH THE OPERATING SYSTEM**

## 4.4.1.1 The os module: Introduction to the os module

In this section, you'll learn about a module called os, which lets you **interact with the operating system using Python.** It provides functions that are available on Unix and/or Windows systems. If you're familiar with the command console, you'll see that some functions give the same results as the commands available on the operating systems.

A good example of this is the *mkdir* function, which allows you to create a directory just like the mkdir command in Unix and Windows. If you don't know this command, don't worry.

You'll soon have the opportunity to learn the functions of the os module, to perform operations on files and directories along with the corresponding commands. In addition to file and directory operations, the os module enables you to:

* get information about the operating system;
* manage processes;
* operate on I/O streams using file descriptors.

In a moment, you'll see how to get basic information about your operating system, although process management and working with file descriptors won't be discussed here, because these are more advanced topics that require knowledge of operating system mechanisms. Ready?

## 4.4.1.2 The os module: Getting information about the operating system

Before you create your first directory structure, you'll see how you can get information about the current operating system. This is really easy because the os module provides a function called *uname*, which returns an object containing the following attributes:

* **systemname** — stores the name of the operating system;
* **nodename** — stores the machine name on the network;
* **release** — stores the operating system release;
* **version** — stores the operating system version;
* **machine** — stores the hardware identifier, e.g., x86\_64.

Let's look at how it is in practice:

|  |
| --- |
| import os  print(os.uname()) |

Result:

*posix.uname\_result(sysname='Linux', nodename='192d19f04766', release='4.4.0-164-generic', version='#192-Ubuntu SMP Fri Sep 13 12:02:50 UTC 2019', machine='x86\_64')*

As you can see, the uname function returns an object containing information about the operating system. The above code was launched on Ubuntu 16.04.6 LTS, so don't be surprised if you get a different result, because it depends on your operating system. Unfortunately, the uname function only works on some Unix systems. If you use Windows, you can use the uname function in the platform module, which returns a similar result.

The os module allows you to quickly distinguish the operating system using the name attribute, which supports one of the following names:

* posix — you'll get this name if you use Unix;
* nt — you'll get this name if you use Windows;
* java — you'll get this name if your code is written in Jython.

For Ubuntu 16.04.6 LTS, the name attribute returns the name posix:

import os

print(os.name)

Result: posix

**NOTE**: On Unix systems, there's a command called uname that returns the same information (if you run it with the -a option) as the uname function.

## 4.4.1.3 The os module: Creating directories in Python

The os module provides a function called *mkdir*, which, like the *mkdir* command in Unix and Windows, allows you to create a directory. The *mkdir* function requires a path that can be relative or absolute. Let's recall what both paths look like in practice:

* my\_first\_directory — this is a relative path which will create the my\_first\_directory directory in the current working directory;
* ./my\_first\_directory — this is a relative path that explicitly points to the current working directory. It has the same effect as the path above;
* ../my\_first\_directory — this is a relative path that will create the my\_first\_directory directory in the parent directory of the current working directory;
* /python/my\_first\_directory — this is the absolute path that will create the my\_first\_directory directory, which in turn is in the python directory in the root directory.

Look at the code in the editor. It shows an example of how to create the my\_first\_directory directory using a relative path. This is the simplest variant of the relative path, which consists of passing only the directory name.

|  |
| --- |
| import os  os.mkdir("my\_first\_directory")  print(os.listdir()) |

If you test your code here, it will output the newly created ['my\_first\_directory'] directory (and the entire content of the current working catalog).

The mkdir function creates a directory in the specified path. Note that running the program twice will raise a FileExistsError.

This means that we cannot create a directory if it already exists. In addition to the path argument, the mkdir function can optionally take the mode argument, which specifies directory permissions. However, on some systems, the mode argument is ignored.

To change the directory permissions, we recommend the *chmod* function, which works similarly to the *chmod* command on Unix systems. You can find more information about it in the documentation. In the above example, another function provided by the os module named *listdir* is used. The listdir function returns a list containing the names of the files and directories that are in the path passed as an argument.

If no argument is passed to it, the current working directory will be used (as in the example above). It's important that the result of the listdir function omits the entries '.' and '..', which are displayed, e.g., when using the ls -a command on Unix systems.

**NOTE**: In both Windows and Unix, there's a command called mkdir, which requires a directory path. The equivalent of the above code that creates the my\_first\_directory directory is the mkdir my\_first\_directory command.

## 4.4.1.4 The os module: Recursive directory creation

The *mkdir* function is very useful, but what if you need to create another directory in the directory you've just created. Of course, you can go to the created directory and create another directory inside it, but fortunately the os module provides a function called ***makedirs***, which makes this task easier. The *makedirs* function enables recursive directory creation, which means that all directories in the path will be created. Let's look at the code in the editor and see how it is in practice.

|  |
| --- |
| import os  os.makedirs("my\_first\_directory/my\_second\_directory")  os.chdir("my\_first\_directory")  print(os.listdir()) |

The code should produce the following result:

['my\_second\_directory']

The code creates two directories. The first of them is created in the current working directory, while the second in the my\_first\_directory directory.

You don't have to go to the my\_first\_directory directory to create the my\_second\_directory directory, because the makedirs function does this for you. In the example above, we go to the my\_first\_directory directory to show that the makedirs command creates the my\_second\_directory subdirectory.

To move between directories, you can use a function called chdir, which changes the current working directory to the specified path. As an argument, it takes any relative or absolute path. In our example, we pass the first directory name to it.

**NOTE**: The equivalent of the makedirs function on Unix systems is the mkdir command with the -p flag, while in Windows, simply the mkdir command with the path:

* Unix-like systems:

mkdir -p my\_first\_directory/my\_second\_directory

* Windows:

mkdir my\_first\_directory/my\_second\_directory

## 4.4.1.5 The os module: Where am I now?

You already know how to create directories and how to move between them. Sometimes, when you have a really large directory structure that you navigate, you may not know which directory you're currently working in.

As you’ve probably guessed, the os module provides a function that returns information about the current working directory. It's called *getcwd*. Look at the code in the editor to see how to use it in practice.

|  |
| --- |
| import os  os.makedirs("my\_first\_directory/my\_second\_directory")  os.chdir("my\_first\_directory")  print(os.getcwd())  os.chdir("my\_second\_directory")  print(os.getcwd()) |

Result:

.../my\_first\_directory

.../my\_first\_directory/my\_second\_directory

In the example, we create the my\_first\_directory directory, and the my\_second\_directory directory inside it. In the next step, we change the current working directory to the my\_first\_directory directory, and then display the current working directory (first line of the result). Next, we go to the my\_second\_directory directory and again display the current working directory (second line of the result). As you can see, the getcwd function returns the absolute path to the directories.

**NOTE**: On Unix-like systems, the equivalent of the *getcwd* function is the pwd command, which prints the name of the current working directory.

## 4.4.1.6 The os module: Deleting directories in Python

The os module also allows you to delete directories. It gives you the option of deleting a single directory or a directory with its subdirectories. To delete a single directory, you can use a function called *rmdir*, which takes the path as its argument. Look at the code in the editor.

|  |
| --- |
| import os  os.mkdir("my\_first\_directory")  print(os.listdir())  os.rmdir("my\_first\_directory")  print(os.listdir()) |

The above example is really simple. First, the my\_first\_directory directory is created, and then it's removed using the rmdir function. The listdir function is used as proof that the directory has been removed successfully. In this case, it returns an empty list. When deleting a directory, make sure it exists and is empty, otherwise an exception will be raised.

To remove a directory and its subdirectories, you can use the *removedirs* function, which requires you to specify a path containing all directories that should be removed:

|  |
| --- |
| import os  os.makedirs("my\_first\_directory/my\_second\_directory")  os.removedirs("my\_first\_directory/my\_second\_directory")  print(os.listdir()) |

As with the *rmdir* function, if one of the directories doesn't exist or isn't empty, an exception will be raised.

**NOTE**: In both Windows and Unix, there's a command called *rmdir*, which, just like the rmdir function, removes directories. What's more, both systems have commands to delete a directory and its contents. In Unix, this is the rm command with the -r flag.

## 4.4.1.7 The os module: The system() function

All functions presented in this part of the course can be replaced by a function called *system*, which executes a command passed to it as a string. The system function is available in both Windows and Unix. Depending on the system, it returns a different result. In Windows, it returns the value returned by the shell after running the command given, while in Unix, it returns the exit status of the process. Let's look at the code in the editor and see how it is in practice.

|  |
| --- |
| import os  returned\_value = os.system("mkdir my\_first\_directory")  print(returned\_value) |

Result: 0

The above example will work in both Windows and Unix. In our case, we receive exit status 0, which indicates success on Unix systems. This means that the my\_first\_directory directory has been created. As part of the exercise, try to list the contents of the directory where you created the my\_first\_directory directory.

## 4.4.1.8 The os module: LAB

Estimated time: 15-30 min

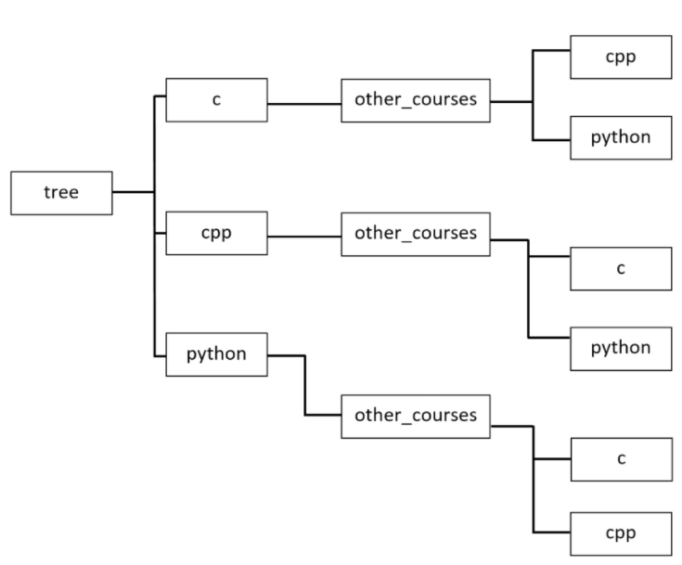
Level of difficulty: Easy

**Objectives**

* improving the student's skills in interacting with the operating system;
* practical use of known functions provided by the os module.

**Scenario**

It goes without saying that operating systems allow you to search for files and directories. While studying this part of the course, you learned about the functions of the os module, which have everything you need to write a program that will search for directories in a given location. To make your task easier, we have prepared a test directory structure for you:



Your program should meet the following requirements:

1. Write a function or method called find that takes two arguments called path and dir. The path argument should accept a relative or absolute path to a directory where the search should start, while the dir argument should be the name of a directory that you want to find in the given path. Your program should display the absolute paths if it finds a directory with the given name.
2. The directory search should be done recursively. This means that the search should also include all subdirectories in the given path.

Example input:

path="./tree", dir="python"

Example output:

.../tree/python

.../tree/cpp/other\_courses/python

.../tree/c/other\_courses/python

## 4.4.1.9 SECTION SUMMARY: Key takeaways

1. The uname function returns an object that contains information about the current operating system. The object has the following attributes:

* **systemname** (stores the name of the operating system)
* **nodename** (stores the machine name on the network)
* **release** (stores the operating system release)
* **version** (stores the operating system version)
* **machine** (stores the hardware identifier, e.g. x86\_64.)

2. The name attribute available in the os module allows you to distinguish the operating system. It returns one of the following three values:

* posix (you'll get this name if you use Unix)
* nt (you'll get this name if you use Windows)
* java (you'll get this name if your code is written in something like Jython)

3. The *mkdir* function creates a directory in the path passed as its argument. The path can be either relative or absolute, e.g:

|  |
| --- |
| import os  os.mkdir("hello") # the relative path  os.mkdir("/home/python/hello") # the absolute path |

**Note**: If the directory exists, a FileExistsError exception will be thrown. In addition to the mkdir function, the os module provides the makedirs function, which allows you to recursively create all directories in a path.

4. The result of the listdir() function is a list containing the names of the files and directories that are in the path passed as its argument.

It's important to remember that the listdir function omits the entries '.' and '..', which are displayed, for example, when using the ls -a command on Unix systems. If the path isn't passed, the result will be returned for the current working directory.

5. To move between directories, you can use a function called chdir(), which changes the current working directory to the specified path. As its argument, it takes any relative or absolute path.

If you want to find out what the current working directory is, you can use the getcwd() function, which returns the path to it.

6. To remove a directory, you can use the rmdir() function, but to remove a directory and its subdirectories, use the removedirs() function.

7. On both Unix and Windows, you can use the system function, which executes a command passed to it as a string, e.g.:

|  |
| --- |
| import os  returned\_value = os.system("mkdir hello") |

The system function on Windows returns the value returned by shell after running the command given, while on Unix it returns the exit status of the process.

**Exercise 1**: What is the output of the following snippet if you run it on Unix?

|  |
| --- |
| import os  print(os.name) |

Check: posix

**Exercise 2**:What is the output of the following snippet?

|  |
| --- |
| import os  os.mkdir("hello")  print(os.listdir()) |

Check: ['hello']

# **4.5. THE DATETIME AND TIME MODULES - WORKING WITH DATE- AND TIME-RELATED FUNCTIONS**

## 4.5.1.1 The datetime module: Introduction to the datetime module 1/21/22

In this section, you'll learn about a Python module called *datetime*. As you can guess, it provides **classes for working with date and time**. If you think you don't need to delve into this topic, let's talk about examples of using date and time in programming.

Date and time have countless uses and it's probably hard to find a production application that doesn't use them. Here are some examples:

* **event logging** — thanks to the knowledge of date and time, we are able to determine when exactly a critical error occurs in our application. When creating logs, you can specify the date and time format;
* **tracking changes in the database** — sometimes it's necessary to store information about when a record was created or modified. The datetime module will be perfect for this case;
* **data validation** — you'll soon learn how to read the current date and time in Python. Knowing the current date and time, we're able to validate various types of data, e.g., whether a discount coupon entered by a user in our application is still valid;
* **storing important information** — can you imagine bank transfers without storing the information of when they were made? The date and time of certain actions must be preserved, and we must deal with it.

Date and time are used in almost every area of our lives, so it's important to familiarize yourself with the Python datetime module. Are you ready for a new dose of knowledge?

## 4.5.1.2 The datetime module: Getting the current local date and creating date objects

One of the classes provided by the datetime module is a class called date. Objects of this class represent a date consisting of the year, month, and day. Look at the code in the editor to see what it looks like in practice and get the current local date using the today method.

Run the code to see what happens.

|  |
| --- |
| from datetime import date  today = date.today()  print("Today:", today) # Today: 2022-01-22  print("Year:", today.year) # Year: 2022  print("Month:", today.month) # Month: 1  print("Day:", today.day) # Day: 22 |

The today method returns a date object representing the current local date. Note that the date object has three attributes: year, month, and day.

Be careful, because these attributes are read-only. To create a date object, you must pass the year, month, and day parameters as follows:

|  |
| --- |
| from datetime import date  my\_date = date(2019, 11, 4)  print(my\_date) # 2019-11-04 |

Run the example to see what happens.

When creating a date object, keep the following restrictions in mind:

|  |  |
| --- | --- |
| Parameter | Restrictions |
| year | The year parameter must be greater than or equal to 1 (MINYEAR constant) and less than or equal to 9999 (MAXYEAR constant). |
| month | month parameter must be greater than or equal to 1 and less than or equal to 12. |
| day | The day parameter must be greater than or equal to 1 and less than or equal to the last day of the given month and year. |

**Note**: Later in this course you'll learn how to change the default date format.

## 4.5.1.3 The datetime module: Creating a date object from a timestamp

The *date* class gives us the ability to create a *date* object from a *timestamp*. In Unix, the timestamp expresses the number of seconds since January 1, 1970, 00:00:00 (UTC). This date is called the **Unix epoch(纪元)**, because this is when the counting of time began on Unix systems. The timestamp is actually the difference between a particular date (including time) and January 1, 1970, 00:00:00 (UTC), expressed in seconds.

To create a date object from a timestamp, we must pass a Unix timestamp to the *fromtimestamp* method. For this purpose, we can use the *time* module, which provides time-related functions. One of them is a function called *time*() that returns the number of seconds from January 1, 1970 to the current moment in the form of a float number. Take a look at the example in the editor.

|  |
| --- |
| from datetime import date  import time  timestamp = time.time()  print("Timestamp:", timestamp)  d = date.fromtimestamp(timestamp)  print("Date:", d) |

Run the code to see the output.

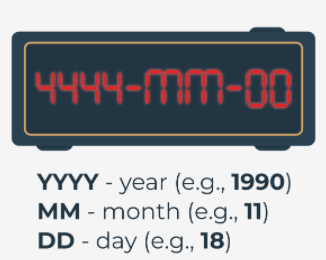
|  |
| --- |
| Timestamp: 1642820070.7663174  Date: 2022-01-22 |

If you run the sample code several times, you'll be able to see how the timestamp increments itself. It's worth adding that the result of the time function depends on the platform, because in Unix and Windows systems, leap seconds aren't counted.

**Note**: In this part of the course we'll also talk about the time module.

## 4.5.1.4 The datetime module: Creating a date object using the ISO format

The *datetime* module provides several methods to create a date object. One of them is the *fromisoformat* method, which takes a date in the **YYYY-MM-DD** format compliant with the ISO 8601 standard. The ISO 8601 standard defines how the date and time are represented. It's often used, so it's worth taking a moment to familiarize yourself with it. Take a look at the picture describing the values required by the format:

Now look at the code in the editor and run it.

|  |
| --- |
| from datetime import date  d = date.fromisoformat('2019-11-04')  print(d) |

In our example, YYYY is 2019, MM is 11 (November), and DD is 04 (fourth day of November).When substituting the date, be sure to add 0 before a month or a day that is expressed by a number less than 10.

**Note**: The fromisoformat method has been available in Python since version 3.7.

## 4.5.1.5 The datetime module: The replace() method

Sometimes you may need to replace the year, month, or day with a different value. You can’t do this with the year, month, and day attributes because they're read-only. In this case, you can use the method named replace. Run the code in the editor.

|  |
| --- |
| **from** datetime **import** date  d **=** date**(**1991**,** 2**,** 5**)**  **print(**d**) #** 1991-02-05  d **=** d**.**replace**(**year**=**1992**,** month**=**1**,** day**=**16**)**  **print(**d**) #** 1992-01-16 |

The year, month, and day parameters are optional. You can pass only one parameter to the replace method, e.g., year, or all three as in the example. The replace method returns a changed date object, so you must remember to assign it to some variable.

## 4.5.1.6 The datetime module: What day of the week is it?

One of the more helpful methods that makes working with dates easier is the method called *weekday*. It returns the day of the week as an integer, where 0 is Monday and 6 is Sunday. Run the code in the editor.

|  |
| --- |
| **from** datetime **import** date  d **=** date**(**2019**,** 11**,** 4**)**  **print(**d**.**weekday**())** # output: 0 |

The date class has a similar method called isoweekday, which also returns the day of the week as an integer, but 1 is Monday, and 7 is Sunday:

|  |
| --- |
| **from** datetime **import** date  d **=** date**(**2019**,** 11**,** 4**)**  **print(**d**.**isoweekday**())** # output: 1 |

As you can see, for the same date we get a different integer, but expressing the same day of the week. The integer returned by the isodayweek method follows the ISO specification.

## 4.5.1.7 The datetime module: Creating time objects

You already know how to present a date using the date object. The datetime module also has a class that allows you to present time. Can you guess its name? Yes, it's called time:

time(hour, minute, second, microsecond, tzinfo, fold)

The time class constructor accepts the following optional parameters:

|  |  |
| --- | --- |
| Parameter | Restrictions |
| hour | The hour parameter must be greater than or equal to 0 and less than 23. |
| minute | The minute parameter must be greater than or equal to 0 and less than 59. |
| second | The second parameter must be greater than or equal to 0 and less than 59. |
| microsecond | The microsecond parameter must be greater than or equal to 0 and less than 1000000. |
| tzinfo | The tzinfo parameter must be a tzinfo subclass object or None (default). |
| fold | The fold parameter must be 0 or 1 (default 0). |

The tzinfo parameter is associated with time zones, while fold with wall times. We won't use them during this course, but we encourage you to familiarize yourself with them. Let's look at how to create a time object in practice. Run the code in the editor.

|  |
| --- |
| **from** datetime **import** time  t **=** time**(**14**,** 53**,** 20**,** 1**)**  **print(**"Time:"**,** t**)**  **print(**"Hour:"**,** t**.**hour**)**  **print(**"Minute:"**,** t**.**minute**)**  **print(**"Second:"**,** t**.**second**)**  **print(**"Microsecond:"**,** t**.**microsecond**)** |

Result:

|  |
| --- |
| Time: 14:53:20.000001  Hour: 14  Minute: 53  Second: 20  Microsecond: 1 |

In the example, we passed four parameters to the class constructor: hour, minute, second, and microsecond. Each of them can be accessed using the class attributes.

**Note**: Soon we'll tell you how you can change the default time formatting.

## 4.5.1.8 The datetime module: The time module

In addition to the time class, the Python standard library offers a module called *time*, which provides a time-related function. You already had the opportunity to learn the function called time when discussing the date class. Now we'll look at another useful function available in this module.

You must spend many hours in front of a computer while doing this course. Sometimes you may feel the need to take a nap. Why not? Let's write a program that simulates a student's short nap. Have a look at the code in the editor.

|  |
| --- |
| **import** time  **class** **Student:**  **def** take\_nap**(**self**,** seconds**):**  **print(**"I'm very tired. I have to take a nap. See you later."**)**  time**.**sleep**(**seconds**)**  **print(**"I slept well! I feel great!"**)**  student **=** Student**()**  student**.**take\_nap**(**5**)** |

Result:

|  |
| --- |
| I'm very tired. I have to take a nap. See you later.  I slept well! I feel great! |

The most important part of the sample code is the use of the sleep function (yes, you may remember it from one of the previous labs earlier in the course), which suspends program execution for the given number of seconds. In our example it's 5 seconds. You're right, it's a very short nap. Extend the student's sleep by changing the number of seconds. Note that the sleep function accepts only an integer or a floating point number.

## 4.5.1.9 The datetime module: The ctime() function

The time module provides a function called ctime, which **converts the time in seconds since January 1, 1970 (Unix epoch) to a string.** Do you remember the result of the time function? That's what you need to pass to ctime. Take a look at the example in the editor.

|  |
| --- |
| **import** time  timestamp **=** 1572879180  **print(**time**.**ctime**(**timestamp**))** |

Result: Mon Nov 4 14:53:00 2019

The ctime function returns a string for the passed timestamp. In our example, the timestamp expresses November 4, 2019 at 14:53:00.

It's also possible to call the ctime function without specifying the time in seconds. In this case, the current time will be returned:

|  |
| --- |
| **import** time  **print(**time**.**ctime**())** |

## 4.5.1.10 The datetime module: The gmtime() and localtime() functions

Some of the functions available in the time module require knowledge of the *struct\_time* class, but before we get to know them, let's see what the class looks like:

|  |
| --- |
| time**.**struct\_time**:**  tm\_year # specifies the year  tm\_mon # specifies the month (value from 1 to 12)  tm\_mday # specifies the day of the month (value from 1 to 31)  tm\_hour # specifies the hour (value from 0 to 23)  tm\_min # specifies the minute (value from 0 to 59)  tm\_sec # specifies the second (value from 0 to 61 )  tm\_wday # specifies the weekday (value from 0 to 6)  tm\_yday # specifies the year day (value from 1 to 366)  tm\_isdst # specifies whether daylight saving time applies (1 – yes, 0 – no, -1 – it isn't known)  tm\_zone # specifies the timezone name (value in an abbreviated form)  tm\_gmtoff # specifies the offset east of UTC (value in seconds) |

The struct\_time class also allows access to values using indexes. Index 0 returns the value in tm\_year, while 8 returns the value in tm\_isdst.

The exceptions are tm\_zone and tm\_gmoff, which cannot be accessed using indexes. Let's look at how to use the struct\_time class in practice. Run the code in the editor.

|  |
| --- |
| **import** time  timestamp **=** 1572879180  **print(**time**.**gmtime**(**timestamp**))**  **print(**time**.**localtime**(**timestamp**))** |

Result:

|  |
| --- |
| time.struct\_time(tm\_year=2019, tm\_mon=11, tm\_mday=4, tm\_hour=14, tm\_min=53, tm\_sec=0, tm\_wday=0, tm\_yday=308, tm\_isdst=0)  time.struct\_time(tm\_year=2019, tm\_mon=11, tm\_mday=4, tm\_hour=14, tm\_min=53, tm\_sec=0, tm\_wday=0, tm\_yday=308, tm\_isdst=0) |

The example shows two functions that convert the elapsed time from the Unix epoch to the struct\_time object. The difference between them is that the gmtime function returns the struct\_time object in UTC, while the localtime function returns local time. For the gmtime function, the tm\_isdst attribute is always 0.

## 4.5.1.11 The datetime module: The asctime() and mktime() functions

The time module has functions that expect a struct\_time object or a tuple that stores values according to the indexes presented when discussing the struct\_time class. Run the example in the editor.

|  |
| --- |
| **import** time  timestamp **=** 1572879180  st **=** time**.**gmtime**(**timestamp**)**  **print(**time**.**asctime**(**st**))**  **print(**time**.**mktime**((**2019**,** 11**,** 4**,** 14**,** 53**,** 0**,** 0**,** 308**,** 0**)))** |

Result:

|  |
| --- |
| Mon Nov 4 14:53:00 2019  1572879180.0 |

The first of the functions, called asctime, converts a struct\_time object or a tuple to a string. Note that the familiar gmtime function is used to get the struct\_time object. If you don't provide an argument to the asctime function, the time returned by the localtime function will be used.

The second function called mktime converts a struct\_time object or a tuple that expresses the local time to the number of seconds since the Unix epoch. In our example, we passed a tuple to it, which consists of the following values:

2019 => tm\_year

11 => tm\_mon

4 => tm\_mday

14 => tm\_hour

53 => tm\_min

0 => tm\_sec

0 => tm\_wday

308 => tm\_yday

0 => tm\_isdst

## 4.5.1.12 The datetime and time modules: Creating datetime objects

In the datetime module, date and time can be represented as separate objects or as one. The class that combines date and time is called datetime.

datetime(year, month, day, hour, minute, second, microsecond, tzinfo, fold)

Its constructor accepts the following parameters:

|  |  |
| --- | --- |
| Parameter | Restrictions |
| year | The year parameter must be greater than or equal to 1 (MINYEAR constant) and less than or equal to 9999 (MAXYEAR constant). |
| month | The month parameter must be greater than or equal to 1 and less than or equal to 12. |
| day | The day parameter must be greater than or equal to 1 and less than or equal to the last day of the given month and year. |
| hour | The hour parameter must be greater than or equal to 0 and less than 23. |
| minute | The minute parameter must be greater than or equal to 0 and less than 59. |
| second | The second parameter must be greater than or equal to 0 and less than 59. |
| microsecond | The microsecond parameter must be greater than or equal to 0 and less than 1000000. |
| tzinfo | The tzinfo parameter must be a tzinfo subclass object or None (default). |
| fold | The fold parameter must be 0 or 1 (default 0). |

Now let's have a look at the code in the editor to see how we create a datetime object.

|  |
| --- |
| **from** datetime **import** datetime  dt **=** datetime**(**2019**,** 11**,** 4**,** 14**,** 53**)**  **print(**"Datetime:"**,** dt**)**  **print(**"Date:"**,** dt**.**date**())**  **print(**"Time:"**,** dt**.**time**())** |

Result:

|  |
| --- |
| Datetime: 2019-11-04 14:53:00  Date: 2019-11-04  Time: 14:53:00 |

The example creates a datetime object representing November 4, 2019 at 14:53:00. All parameters passed to the constructor go to read-only class attributes. They're year, month, day, hour, minute, second, microsecond, tzinfo, and fold. The example shows two methods that return two different objects. The method called date returns the date object with the given year, month, and day, while the method called time returns the time object with the given hour and minute.

## 4.5.1.13 The datetime and time modules: Methods that return the current date and time

The datetime class has several methods returning the current date and time, including:

* today() — returns the current local date and time with the tzinfo attribute set to None;
* now() — returns the current local date and time the same as the today method, unless we pass the optional argument tz to it. The argument of this method must be an object of the tzinfo subclass;
* utcnow() — returns the current UTC date and time with the tzinfo attribute set to None.

Run the code in the editor to see them all in practice. What can you say about the output?

|  |
| --- |
| **from** datetime **import** datetime  **print(**"today:"**,** datetime**.**today**())**  **print(**"now:"**,** datetime**.**now**())**  **print(**"utcnow:"**,** datetime**.**utcnow**())** |

Results:

|  |
| --- |
| today: 2022-01-22 04:19:24.488809  now: 2022-01-22 04:19:24.493387  utcnow: 2022-01-22 04:19:24.493418 |

As you can see, the result of all the three methods is the same. The small differences are caused by the time elapsed between subsequent calls.

**Note**: You can read more about tzinfo objects in the documentation.

## 4.5.1.14 The datetime and time modules: Getting a timestamp 1/22/22

There are many converters available on the Internet that can calculate a timestamp based on a given date and time, but how can we do it in the datetime module? This is possible thanks to the timestamp method provided by the datetime class. Look at the code in the editor.

|  |
| --- |
| **from** datetime **import** datetime  dt **=** datetime**(**2020**,** 10**,** 4**,** 14**,** 55**)**  **print(**"Timestamp:"**,** dt**.**timestamp**())** |

Result: Timestamp: 1601823300.0

The *timestamp* method returns a float value expressing the number of seconds elapsed between date and time indicated by the datetime object and January 1, 1970, 00:00:00 (UTC).

## 4.5.1.15 The datetime and time modules: Date and time formatting (part 1)

All datetime module classes presented so far have a method called *strftime*. This is a very important method, because it allows us to return the date and time in the format we specify. The *strftime* method takes only one argument in the form of a string specifying the format that can consist of directives. A directive is a string consisting of the character % (percent) and a lowercase or uppercase letter, e.g., the directive %Y means the year with the century as a decimal number. Let's see it in an example. Run the code in the editor.

|  |
| --- |
| **from** datetime **import** date  d **=** date**(**2020**,** 1**,** 4**)**  **print(**d**.**strftime**(**'%Y/%m/%d'**))** |

Result:2020/01/04

In the example, we passed a format consisting of three directives separated by / (slash) to the strftime method. Of course, the separator character can be replaced by another character, or even by a string. You can put any characters in the format, but only recognizable directives will be replaced with the appropriate values. In our format we've used the directives:

* %Y – returns the year with the century as a decimal number. In example, this is 2020.
* %m – returns the month as a zero-padded decimal number. In our example, it's 01.
* %d – returns the day as a zero-padded decimal number. In our example, it's 04.

Note: You can find all available directives here.([datetime — Basic date and time types — Python 3.10.2 documentation](https://docs.python.org/3/library/datetime.html" \l "strftime-and-strptime-format-codes))

## 4.5.1.16 The datetime and time modules: Date and time formatting (part 2)

Time formatting works in the same way as date formatting, but requires the use of appropriate directives. Let's take a closer look at a few of them in the editor.

|  |
| --- |
| **from** datetime **import** time  **from** datetime **import** datetime  t **=** time**(**14**,** 53**)**  **print(**t**.**strftime**(**"%H:%M:%S"**))**  dt **=** datetime**(**2020**,** 11**,** 4**,** 14**,** 53**)**  **print(**dt**.**strftime**(**"%y/%B/%d %H:%M:%S"**))** |

Result:

|  |
| --- |
| 14:53:00  20/November/04 14:53:00 |

The first of the formats used concerns only time. As you can guess, %H returns the hour as a zero-padded decimal number, %M returns the minute as a zero-padded decimal number, while %S returns the second as a zero-padded decimal number. In our example, %H is replaced by 14, %M by 53, and %S by 00.

The second format used combines date and time directives. There are two new directives, %Y and %B. The directive %Y returns the year without a century as a zero-padded decimal number (in our example it's 20). The %B directive returns the month as the locale’s full name (in our example, it's November).

In general, you've got a lot of freedom in creating formats, but you must remember to use the directives properly. As an exercise, you can check what happens if, for example, you try to use the %Y directive in the format passed to the time object's strftime method. Try to find out why you got this result yourself. Good luck!

## 4.5.1.17 The datetime and time modules: The strftime() function in the time module

You probably won't be surprised to learn that the *strftime* function is available in the time module. It differs slightly from the *strftime* methods in the classes provided by the datetime module because, in addition to the format argument, it can also take (optionally) a tuple or struct\_time object. If you don't pass a tuple or struct\_time object, the formatting will be done using the current local time. Take a look at the example in the editor.

|  |
| --- |
| **import** time  timestamp **=** 1572879180  st **=** time**.**gmtime**(**timestamp**)**  **print(**time**.**strftime**(**"%Y/%m/%d %H:%M:%S"**,** st**))**  **print(**time**.**strftime**(**"%Y/%m/%d %H:%M:%S"**))** |

Our result looks as follows:

|  |
| --- |
| 2019/11/04 14:53:00  2020/10/12 12:19:40 |

Creating a format looks the same as for the strftime methods in the datetime module. In our example, we use the %Y, %m, %d, %H, %M, and %S directives that you already know.

In the first function call, we format the struct\_time object, while in the second call (without the optional argument), we format the local time. You can find all available directives in the time module here.([time — Time access and conversions — Python 3.10.2 documentation](https://docs.python.org/3/library/time.html" \l "time.strftime))

## 4.5.1.18 The datetime and time modules: The strptime() method

Knowing how to create a format can be helpful when using a method called *strptime* in the datetime class. Unlike the *strftime* method, it creates a datetime object from a string representing a date and time. The *strptime* method requires you to specify the format in which you saved the date and time. Let's see it in an example. Take a look at the code in the editor.

|  |
| --- |
| **from** datetime **import** datetime  **print(**datetime**.**strptime**(**"2019/11/04 14:53:00"**,** "%Y/%m/%d %H:%M:%S"**))** |

Result: 2019-11-04 14:53:00

In the example, we've specified two required arguments. The first is a date and time as a string: "2019/11/04 14:53:00", while the second is a format that facilitates parsing to a datetime object. Be careful, because if the format you specify doesn't match the date and time in the string, it'll raise a ValueError.

**Note**: In the time module, you can find a function called strptime, which parses a string representing a time to a struct\_time object. Its use is analogous to the strptime method in the datetime class:

|  |
| --- |
| **import** time  **print(**time**.**strptime**(**"2019/11/04 14:53:00"**,** "%Y/%m/%d %H:%M:%S"**))** |

Its result will be as follows:

|  |
| --- |
| time.struct\_time(tm\_year=2019, tm\_mon=11, tm\_mday=4, tm\_hour=14, tm\_min=53, tm\_sec=0, tm\_wday=0, tm\_yday=308, tm\_isdst=-1) |

## 4.5.1.19 The datetime and time modules: Date and time operations

Sooner or later you'll have to perform some calculations on the date and time. Fortunately, there's a class called *timedelta* in the *datetime* module that was created for just such a purpose.

To create a *timedelta* object, just do subtraction on the *date* or *datetime* objects, just like we did in the example in the editor. Run it.

|  |
| --- |
| **from** datetime **import** date  **from** datetime **import** datetime  d1 **=** date**(**2020**,** 11**,** 4**)**  d2 **=** date**(**2019**,** 11**,** 4**)**  **print(**d1 **-** d2**)**  dt1 **=** datetime**(**2020**,** 11**,** 4**,** 0**,** 0**,** 0**)**  dt2 **=** datetime**(**2019**,** 11**,** 4**,** 14**,** 53**,** 0**)**  **print(**dt1 **-** dt2**)** |

Result:

|  |
| --- |
| 366 days, 0:00:00  365 days, 9:07:00 |

The example shows subtraction for both the *date* and *datetime* objects. In the first case, we receive the difference in days, which is 366 days. Note that the difference in hours, minutes, and seconds is also displayed. In the second case, we receive a different result, because we specified the time that was included in the calculations. As a result, we receive 365 days, 9 hours, and 7 minutes. In a moment you'll learn more about creating timedelta objects and about the operations you can do with them.

## 4.5.1.20 The datetime and time modules: Creating timedelta objects

You've already learned that a *timedelta* object can be returned as a result of subtracting two *date* or *datetime* objects. Of course, you can also create an object yourself. For this purpose, let's get acquainted with the arguments accepted by the class constructor, which are: *days, seconds, microseconds, milliseconds, minutes, hours, and weeks*. Each of them is optional and defaults to 0. The arguments should be integers or floating point numbers, and can be either positive or negative. Let's look at a simple example in the editor.

|  |
| --- |
| **from** datetime **import** timedelta  delta **=** timedelta**(**weeks**=**2**,** days**=**2**,** hours**=**3**)**  **print(**delta**)** |

Result: 16 days, 3:00:00

The result of 16 days is obtained by converting the *weeks* argument to days (2 weeks = 14 days) and adding the *days* argument (2 days). This is normal behavior, because the *timedelta* object only stores days, seconds, and microseconds internally. Similarly, the *hour* argument is converted to minutes. Take a look at the example below:

|  |
| --- |
| **from** datetime **import** timedelta  delta **=** timedelta**(**weeks**=**2**,** days**=**2**,** hours**=**3**)**  **print(**"Days:"**,** delta**.**days**)**  **print(**"Seconds:"**,** delta**.**seconds**)**  **print(**"Microseconds:"**,** delta**.**microseconds**)** |

Result:

|  |
| --- |
| Days: 16  Seconds: 10800  Microseconds: 0 |

The result of 10800 is obtained by converting 3 hours into seconds. In this way the timedelta object stores the arguments passed during its creation. Weeks are converted to days, hours and minutes to seconds, and milliseconds to microseconds.

## 4.5.1.21 The datetime and time modules: Creating timedelta objects: continued

You already know how the *timedelta* object stores the passed arguments internally. It's time to see how it can be used in practice. Let's look at some operations supported by the datetime module classes. Run the code we've provided in the editor.

|  |
| --- |
| **from** datetime **import** timedelta  **from** datetime **import** date  **from** datetime **import** datetime  delta **=** timedelta**(**weeks**=**2**,** days**=**2**,** hours**=**2**)**  **print(**delta**)**  delta2 **=** delta **\*** 2  **print(**delta2**)**  d **=** date**(**2019**,** 10**,** 4**)** **+** delta2  **print(**d**)**  dt **=** datetime**(**2019**,** 10**,** 4**,** 14**,** 53**)** **+** delta2  **print(**dt**)** |

Result:

|  |
| --- |
| 16 days, 2:00:00  32 days, 4:00:00  2019-11-05  2019-11-05 18:53:00 |

The *timedelta* object can be multiplied by an integer. In our example, we multiply the object representing 16 days and 2 hours by 2. As a result, we receive a *timedelta* object representing 32 days and 4 hours.

Note that both days and hours have been multiplied by 2. Another interesting operation using the *timedelta* object is adding. In the example, we've added the *timedelta* object to the date and *datetime* objects.

As a result of these operations, we receive *date* and *datetime* objects increased by days and hours stored in the *timedelta* object.

The presented multiplication operation allows you to quickly increase the value of the *timedelta* object, while multiplication can also help you get a date from the future.

Of course, the *timedelta, date, and datetime* classes support many more operations. We encourage you to familiarize yourself with them in the documentation.

## 4.5.1.22 The datetime and time modules: LAB

Estimated time: 15-45 min

Level of difficulty: Easy

**Objectives**

* improving the student's skills in date and time formatting;
* improving the student's skills in using the *strftime* method.

**Scenario**

During this course, you learned about the *strftime* method, which requires knowledge of directives to create a format. It's time to put the known directives into practice. By the way, you'll have the opportunity to practice working with documentation, because you'll have to find directives that you don't yet know.

**Here's your task:**

Write a program that creates a *datetime* object for November 4, 2020, 14:53:00. The object created should call the strftime method with the appropriate format to display the following result:

|  |
| --- |
| 2020/11/04 14:53:00  20/November/04 14:53:00 PM  Wed, 2020 Nov 04  Wednesday, 2020 November 04  Weekday: 3  Day of the year: 309  Week number of the year: 44 |

**Note**: Each result line should be created by calling the strftime method with at least one directive in the format argument.

## 4.5.1.23 SECTION SUMMARY: Key takeaways

1. To create a *date* object, you must pass the year, month, and day arguments as follows:

|  |
| --- |
| **from** datetime **import** date  my\_date **=** date**(**2020**,** 9**,** 29**)**  **print(**"Year:"**,** my\_date**.**year**)** # Year: 2020  **print(**"Month:"**,** my\_date**.**month**)** # Month: 9  **print(**"Day:"**,** my\_date**.**day**)** # Day: 29 |

The *date* object has three (read-only) attributes: year, month, and day.

2. The *today* method returns a date object representing the current local date:

|  |
| --- |
| **from** datetime **import** date  **print(**"Today:"**,** date**.**today**())** # Displays: Today: 2020-09-29 |

3. In Unix, the timestamp expresses the number of seconds since January 1, 1970, 00:00:00 (UTC). This date is called the "Unix epoch", because it began the counting of time on Unix systems. The timestamp is actually the difference between a particular date (including time) and January 1, 1970, 00:00:00 (UTC), expressed in seconds. To create a date object from a timestamp, we must pass a Unix timestamp to the *fromtimestamp* method:

|  |
| --- |
| **from** datetime **import** date  **import** time  timestamp **=** time**.**time**()**  d **=** date**.**fromtimestamp**(**timestamp**)** |

Note: The *time* function returns the number of seconds from January 1, 1970 to the current moment in the form of a float number.

4. The constructor of the *time* class accepts six arguments (hour, minute, second, microsecond, tzinfo, and fold). Each of these arguments is optional.

|  |
| --- |
| **from** datetime **import** time  t **=** time**(**13**,** 22**,** 20**)**  **print(**"Hour:"**,** t**.**hour**)** # Hour: 13  **print(**"Minute:"**,** t**.**minute**)** # Minute: 22  **print(**"Second:"**,** t**.**second**)** # Second: 20 |

5. The *time* module contains the sleep function, which suspends program execution for a given number of seconds, e.g.:

|  |
| --- |
| **import** time  time**.**sleep**(**10**)**  **print(**"Hello world!"**)** # This text will be displayed after 10 seconds. |

6. In the *datetime* module, date and time can be represented either as separate objects, or as one object. The class that combines date and time is called *datetime*. All arguments passed to the constructor go to read-only class attributes. They are *year, month, day, hour, minute, second, microsecond, tzinfo, and fold*:

|  |
| --- |
| **from** datetime **import** datetime  dt **=** datetime**(**2020**,** 9**,** 29**,** 13**,** 51**)**  **print(**"Datetime:"**,** dt**)** # Displays: Datetime: 2020-09-29 13:51:00 |

7. The *strftime* method takes only one argument in the form of a string specifying a format that can consist of directives. A directive is a string consisting of the character % (percent) and a lowercase or uppercase letter. Below are some useful directives:

* %Y – returns the year with the century as a decimal number;
* %m – returns the month as a zero-padded decimal number;
* %d – returns the day as a zero-padded decimal number;
* %H – returns the hour as a zero-padded decimal number;
* %M – returns the minute as a zero-padded decimal number;
* %S – returns the second as a zero-padded decimal number.

Example:

|  |
| --- |
| **from** datetime **import** date  d **=** date**(**2020**,** 9**,** 29**)**  **print(**d**.**strftime**(**'%Y/%m/%d'**))** # Displays: 2020/09/29 |

8. It's possible to perform calculations on *date* and *datetime* objects, e.g.:

|  |
| --- |
| **from** datetime **import** date  d1 **=** date**(**2020**,** 11**,** 4**)**  d2 **=** date**(**2019**,** 11**,** 4**)**  d **=** d1 **-** d2  **print(**d**)** # Displays: 366 days, 0:00:00.  **print(**d **\*** 2**)** # Displays: 732 days, 0:00:00. |

The result of the subtraction is returned as a *timedelta* object that expresses the difference in days between the two dates in the example above.

Note that the difference in hours, minutes, and seconds is also displayed. The *timedelta* object can be used for further calculations (e.g. you can multiply it by 2).

**Exercise 1**: What is the output of the following snippet?

|  |
| --- |
| **from** datetime **import** time  t **=** time**(**14**,** 39**)**  **print(**t**.**strftime**(**"%H:%M:%S"**))** |

Check: 14:39:00

**Exercise 2**: What is the output of the following snippet?

|  |
| --- |
| **from** datetime **import** datetime  dt1 **=** datetime**(**2020**,** 9**,** 29**,** 14**,** 41**,** 0**)**  dt2 **=** datetime**(**2020**,** 9**,** 28**,** 14**,** 41**,** 0**)**  **print(**dt1 **-** dt2**)** |

Check: 1 day, 0:00:00

# **4.6. The Calendar Module - Working With Calendar - Related Functions**

## 4.6.1.1 The calendar module: Introduction to the calendar module

In addition to the *datetime* and *time* modules, the Python standard library provides a module called *calendar* which, as the name suggests, offers **calendar-related functions**.

One of them is of course displaying the calendar. It's important that the days of the week are displayed from Monday to Sunday, and each day of the week has its representation in the form of an integer:

|  |  |  |
| --- | --- | --- |
| Day of the week | Integer value | Constant |
| Monday | 0 | calendar.MONDAY |
| Tuesday | 1 | calendar.TUESDAY |
| Wednesday | 2 | calendar.WEDNESDAY |
| Thursday | 3 | calendar.THURSDAY |
| Friday | 4 | calendar.FRIDAY |
| Saturday | 5 | calendar.SATURDAY |
| Sunday | 6 | calendar.SUNDAY |

The table above shows the representation of the days of the week in the calendar module. The first day of the week (Monday) is represented by the value 0 and the calendar.MONDAY constant, while the last day of the week (Sunday) is represented by the value 6 and the calendar.SUNDAY constant.

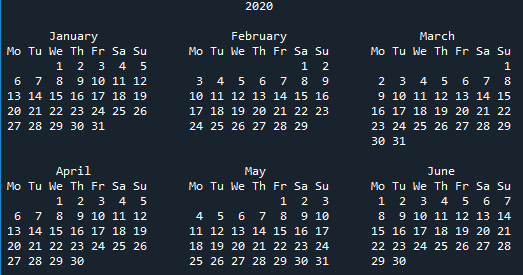
For months, integer values are indexed from 1, i.e., January is represented by 1, and December by 12. Unfortunately, there aren't constants that express the months.

The above information will be useful to you when working with the calendar module in this part of the course, but first let's start with some simple calendar examples.

## 4.6.1.2 The calendar module: Your first calendar

You will start your adventure with the *calendar* module with a simple function called *calendar*, which allows you to **display the calendar for the whole year**. Let's look at how to use it to display the calendar for 2020. Run the code in the editor and see what happens.

|  |
| --- |
| **import** calendar  **print(**calendar**.**calendar**(**2020**))** |

The result displayed is similar to the result of the *cal* command available in Unix. If you want to change the default calendar formatting, you can use the following parameters:

* w – date column width (default 2)
* l – number of lines per week (default 1)
* c – number of spaces between month columns (default 6)
* m – number of columns (default 3)

The calendar function requires you to specify the year, while the other parameters responsible for formatting are optional. We encourage you to try these parameters yourself.

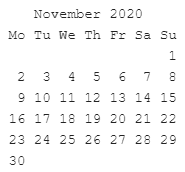
A good alternative to the above function is the function called *prcal*, which also takes the same parameters as the calendar function, but doesn't require the use of the print function to display the calendar. Its use looks like this:

|  |
| --- |
| **import** calendar  calendar**.**prcal**(**2020**)** |

## 4.6.1.3 The calendar module: Calendar for a specific month

The *calendar* module has a function called *month*, which allows you to display a calendar for a specific month. Its use is really simple, you just need to specify the year and month - check out the code in the editor.

|  |
| --- |
| **import** calendar  **print(**calendar**.**month**(**2020**,** 11**))** |

 The example displays the calendar for November 2020. As in the calendar function, you can change the default formatting using the following parameters:

* w – date column width (default 2)
* l – number of lines per week (default 1)

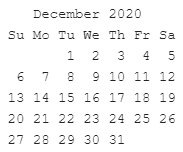
**Note**: You can also use the *prmonth* function, which has the same parameters as the *month* function, but doesn't require you to use the *print* function to display the calendar.

## 4.6.1.4 The calendar module: The setfirstweekday() function

As you already know, by default in the *calendar* module, the first day of the week is Monday. However, you can change this behavior using a function called *setfirstweekday*.

Do you remember the table showing the days of the week and their representation in the form of integer values? It's time to use it, because the *setfirstweekday* method requires a parameter expressing the day of the week in the form of an integer value. Take a look at the example in the editor.

|  |
| --- |
| **import** calendar  calendar**.**setfirstweekday**(**calendar**.**SUNDAY**)**  calendar**.**prmonth**(**2020**,** 12**)** |

 The example uses the *calendar.SUNDAY* constant, which contains a value of 6. Of course, you could pass this value directly to the *setfirstweekday* function, but the version with a constant is more elegant.

As a result, we get a calendar showing the month of December 2020, in which the first day of all the weeks is Sunday.

## 4.6.1.5 The calendar module: The weekday() function

Another useful function provided by the calendar module is the function called *weekday*, which returns the day of the week as an integer value for the given year, month, and day. Let's see it in practice. Run the code in the editor to check the day of the week that falls on December 24, 2020.

|  |
| --- |
| **import** calendar  **print(**calendar**.**weekday**(**2020**,** 12**,** 24**))** |

Result: 3

The weekday function returns 3, which means that December 24, 2020 is a Thursday.

## 4.6.1.6 The calendar module: The weekheader() function

You've probably noticed that the calendar contains weekly headers in a shortened form. If needed, you can get short weekday names using the *weekheader* method.

The *weekheader* method requires you to specify the width in characters for one day of the week. If the width you provide is greater than 3, you'll still get the abbreviated weekday names consisting of three characters. So let's look at how to get a smaller header. Run the code in the editor.

|  |
| --- |
| **import** calendar  **print(**calendar**.**weekheader**(**3**))** |

Result: Mo Tu We Th Fr Sa Su

Note: If you change the first day of the week, e.g., using the setfirstweekday function, it'll affect the result of the weekheader function.

## 4.6.1.7 The calendar module: How do we check if a year is a leap year?

The *calendar* module provides two useful functions to check whether years are leap years.

The first one, called *isleap*, returns True if the passed year is leap, or False otherwise. The second one, called *leapdays*, returns the number of leap years in the given range of years.

Run the code in the editor.

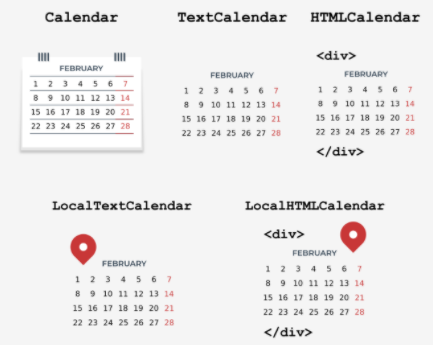
|  |
| --- |
| **import** calendar  **print(**calendar**.**isleap**(**2020**))** # Output True.  **print(**calendar**.**leapdays**(**2010**,** 2021**))** # Up to but not including 2021. # Output 3. |

In the example, we obtain the result 3, because in the period from 2010 to 2020 there are only three leap years (note: 2021 is not included). They are the years 2012, 2016, and 2020.

## 4.6.1.8 The calendar module: Classes for creating calendars

The presented functions aren't everything the *calendar* module offers. In addition to them, we can use the following classes:

* *calendar.Calendar* – provides methods to prepare calendar data for formatting;
* *calendar.TextCalendar* – is used to create regular text calendars;
* *calendar.HTMLCalendar* – is used to create HTML calendars;
* *calendar.LocalTextCalendar* – is a subclass of the *calendar.TextCalendar* class. The constructor of this class takes the locale parameter, which is used to return the appropriate months and weekday names.
* *calendar.LocalHTMLCalendar* – is a subclass of the *calendar.HTMLCalendar* class. The constructor of this class takes the locale parameter, which is used to return the appropriate months and weekday names.

During this course, you've already had the opportunity to create text calendars when discussing the functions of the *calendar* module. Time to try something new. Let's take a closer look at the methods of the calendar class.

## 4.6.1.9 The calendar module: Creating a Calendar object 1/23/22

The *Calendar* class constructor takes one optional parameter named firstweekday, by default equal to 0 (Monday). The *firstweekday* parameter must be an integer between 0-6. For this purpose, we can use the already-known constants - look at the code in the editor.

|  |
| --- |
| **import** calendar  c **=** calendar**.**Calendar**(**calendar**.**SUNDAY**)**  **for** weekday **in** c**.**iterweekdays**():**  **print(**weekday**,** end**=**" "**)** |

The program will output the following result: 6 0 1 2 3 4 5

The code example uses the Calendar class method named *iterweekdays*, which returns an iterator for week day numbers. The first value returned is always equal to the value of the *firstweekday* property. Because in our example the first value returned is 6, it means that the week starts on a Sunday.

## 4.6.1.10 The calendar module: The itermonthdates() method

The *Calendar* class has several methods that return an iterator. One of them is the *itermonthdates* method, which requires specifying the year and month. As a result, all days in the specified month and year are returned, as well as all days before the beginning of the month or the end of the month that are necessary to get a complete week. Each day is represented by a *datetime.date* object. Take a look at the example in the editor.

|  |
| --- |
| **import** calendar  c **=** calendar**.**Calendar**()**  **for** date **in** c**.**itermonthdates**(**2019**,** 11**):**  **print(**date**,** end**=**" "**)** |

The code displays all days in November 2019. Because the first day of November 2019 was a Friday, the following days are also returned to get the complete week: 10/28/2019 (Monday)

10/29/2019 (Tuesday) 10/30/2019 (Wednesday) 10/31/2019 (Thursday).

|  |
| --- |
| 2019-10-28 2019-10-29 2019-10-30 2019-10-31 2019-11-01 2019-11-02 2019-11-03 2019-11-04 2019-11-05 2019-11-06 2019-11-07 2019-11-08 2019-11-09 2019-11-10 2019-11-11 2019-11-12 2019-11-13 2019-11-14 2019-11-15 2019-11-16 2019-11-17 2019-11-18 2019-11-19 2019-11-20 2019-11-21 2019-11-22 2019-11-23 2019-11-24 2019-11-25 2019-11-26 2019-11-27 2019-11-28 2019-11-29 2019-11-30 2019-12-01 |

The last day of November 2019 was a Saturday, so in order to keep the complete week, one more day is returned 12/01/2019 (Friday).

## 4.6.1.11 The calendar module: Other methods that return iterators

Another useful method in the Calendar class is the method called *itermonthdates*, which takes year and month as parameters, and then returns the iterator to the days of the week represented by numbers. Take a look at the example in the editor.

|  |
| --- |
| **import** calendar  c **=** calendar**.**Calendar**()**  **for** **iter** **in** c**.**itermonthdays**(**2019**,** 11**):**  **print(iter,** end**=**" "**)** |

You’ll have certainly noticed the large number of 0s returned as a result of the example code. These are days outside the specified month range that are added to keep the complete week.

|  |
| --- |
| 0 0 0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 0 |

The first four zeros represent 10/28/2019 (Monday) 10/29/2019 (Tuesday) 10/30/2019 (Wednesday) 10/31/2019 (Thursday). The remaining numbers are days in the month, except the last value of 0, which replaces the date 12/01/2019 (Sunday).

There are four other similar methods in the Calendar class that differ in data returned:

* *itermonthdates2* – returns days in the form of tuples consisting of a day of the month number and a week day number;
* *itermonthdates3* – returns days in the form of tuples consisting of a year, a month, and a day of the month numbers. This method has been available since version 3.7;
* *itermonthdates4* – returns days in the form of tuples consisting of a year, a month, a day of the month, and a day of the week numbers. This method has been available since Python version 3.7.

For testing purposes, use the example above and see how the return values of the described methods look in practice.

## 4.6.1.12 The calendar module: The monthdays2calendar() method

The *Calendar* class has several other useful methods that you can learn more about in the documentation (https://docs.python.org/3/library/calendar.html).

One of them is the *monthdays2calendar* method, which takes the year and month, and then returns a list of weeks in a specific month. Each week is a tuple consisting of day numbers and weekday numbers. Look at the code in the editor.

|  |
| --- |
| **import** calendar  c **=** calendar**.**Calendar**()**  **for** data **in** c**.**monthdays2calendar**(**2020**,** 12**):**  **print(**data**)** |

Note that the days numbers outside the month are represented by 0, while the weekday numbers are a number from 0-6, where 0 is Monday and 6 is Sunday. In a moment, this method may be useful for you to complete a laboratory task. Are you ready?

## 4.6.1.13 LAB: the calendar module

**LAB**

Estimated time: 30-60 minutes

Level of difficulty: Easy

**Objectives**

Improving the student's skills in using the Calendar class.

**Scenario**

During this course, we looked at the Calendar class a bit. Your task is to extend its functionality with a new method called count\_weekday\_in\_year, which takes a year and a weekday as parameters, and then returns the number of occurrences of a specific weekday in the year.

Use the following tips:

* Create a class called MyCalendar that extends the Calendar class;
* create the count\_weekday\_in\_year method with the year and weekday parameters. The weekday parameter should be a value between 0-6, where 0 is Monday and 6 is Sunday. The method should return the number of days as an integer;
* in your implementation, use the monthdays2calendar method of the Calendar class.

The following are the expected results:

**Sample arguments:** year=2019, weekday=0

Expected output: 52

Sample arguments: year=2000, weekday=6

Expected output: 53

## 4.6.1.14 SECTION SUMMARY: Key takeaways

1. In the *calendar* module, the days of the week are displayed from Monday to Sunday. Each day of the week has its representation in the form of an integer, where the first day of the week (Monday) is represented by the value 0, while the last day of the week (Sunday) is represented by the value 6.

2. To display a calendar for any year, call the *calendar* function with the year passed as its argument, e.g.:

|  |
| --- |
| **import** calendar  **print(**calendar**.**calendar**(**2020**))** |

Note: A good alternative to the above function is the function called *prcal*, which also takes the same parameters as the calendar function, but doesn't require the use of the *print* function to display the calendar.

3. To display a calendar for any month of the year, call the *month* function, passing year and month to it. For example:

|  |
| --- |
| **import** calendar  **print(**calendar**.**month**(**2020**,** 9**))** |

Note: You can also use the *prmonth* function, which has the same parameters as the *month* function, but doesn't require the use of the *print* function to display the calendar.

4. The *setfirstweekday* function allows you to change the first day of the week. It takes a value from 0 to 6, where 0 is Sunday and 6 is Saturday.

5. The result of the weekday function is a day of the week as an integer value for a given year, month, and day:

|  |
| --- |
| **import** calendar  **print(**calendar**.**weekday**(**2020**,** 9**,** 29**))** # This displays 1, which means Tuesday. |

6. The weekheader function returns the weekday names in a shortened form. The weekheader method requires you to specify the width in characters for one day of the week. If the width you provide is greater than 3, you'll still get the abbreviated weekday names consisting of only three characters. For example:

|  |
| --- |
| **import** calendar  **print(**calendar**.**weekheader**(**2**))** # This display: Mo Tu We Th Fr Sa Su |

7. A very useful function available in the calendar module is the function called *isleap*, which, as the name suggests, allows you to check whether the year is a leap year or not:

|  |
| --- |
| **import** calendar  **print(**calendar**.**isleap**(**2020**))** # This displays: True |

8. You can create a *calendar* object yourself using the *Calendar* class, which, when creating its object, allows you to change the first day of the week with the optional *firstweekday* parameter, e.g.:

|  |
| --- |
| **import** calendar  c **=** calendar**.**Calendar**(**2**)**  **for** weekday **in** c**.**iterweekdays**():**  **print(**weekday**,** end**=**" "**)**  # Result: 2 3 4 5 6 0 1 |

The *iterweekdays* returns an iterator for weekday numbers. The first value returned is always equal to the value of the *firstweekday* property.

**Exercise 1**:What is the output of the following snippet?

|  |
| --- |
| **import** calendar  **print(**calendar**.**weekheader**(**1**))**  # Check: M T W T F S S |

**Exercise 2**: What is the output of the following snippet?

|  |
| --- |
| **import** calendar  c **=** calendar**.**Calendar**()**  **for** weekday **in** c**.**iterweekdays**():**  **print(**weekday**,** end**=**" "**)**  #Check: 0 1 2 3 4 5 6 |

## 4.6.1.15 Module Completion:

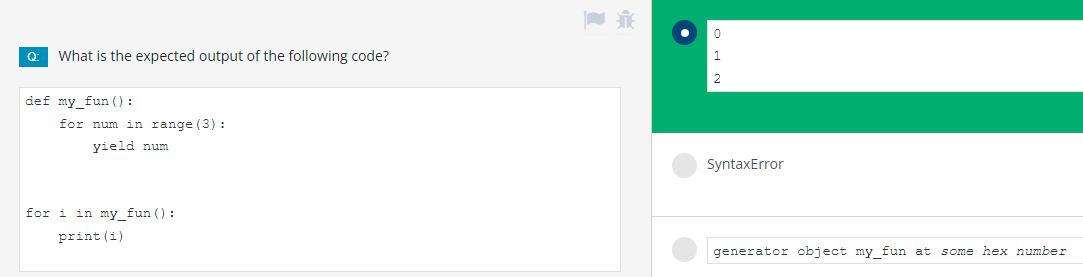
Congratulations! You have completed PE2: Module 4.

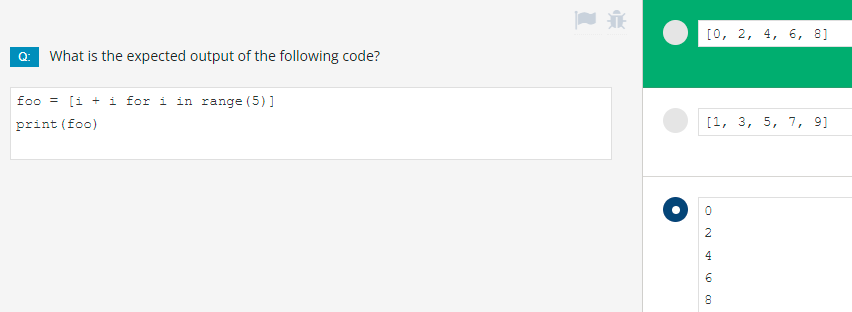
Well done! You've reached the end of Module 4 and completed a major milestone in your Python programming education. Here's a short summary of the objectives you've covered and got familiar with in Module 4:

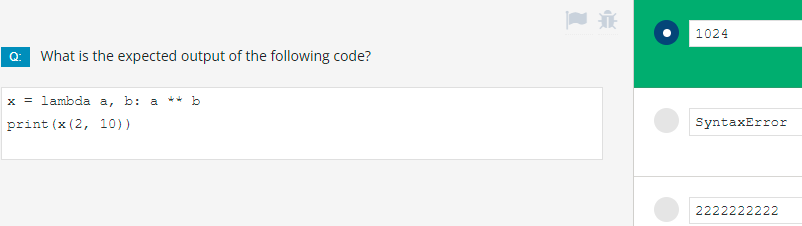
* generators and iterators;
* list comprehensions;
* the lambda, map, and filter functions;
* closures;
* working with files (file streams, file processing, diagnosing stream problems)
* processing text and binary files;
* selected Python STL modules: os, datetime, time, and calendar.

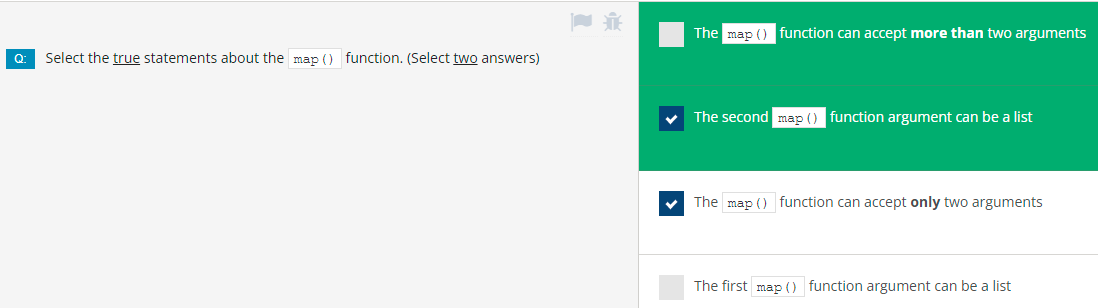
You are now ready to take the module quiz and attempt the final challenge: Module 4 Test, which will help you gauge what you've learned so far.

# 4.7 PE2 -- Module 4 Quiz

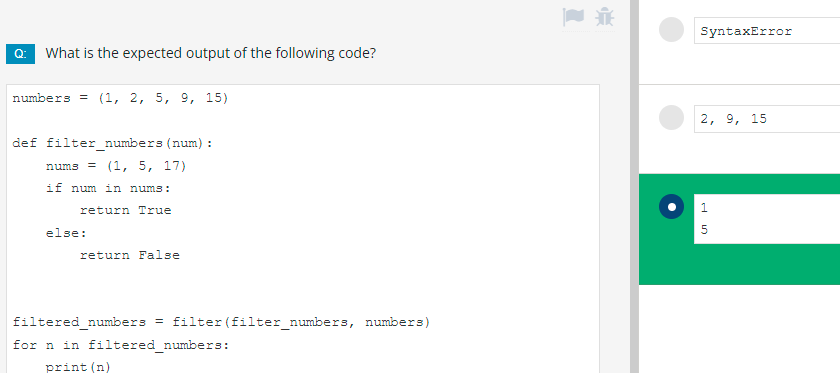


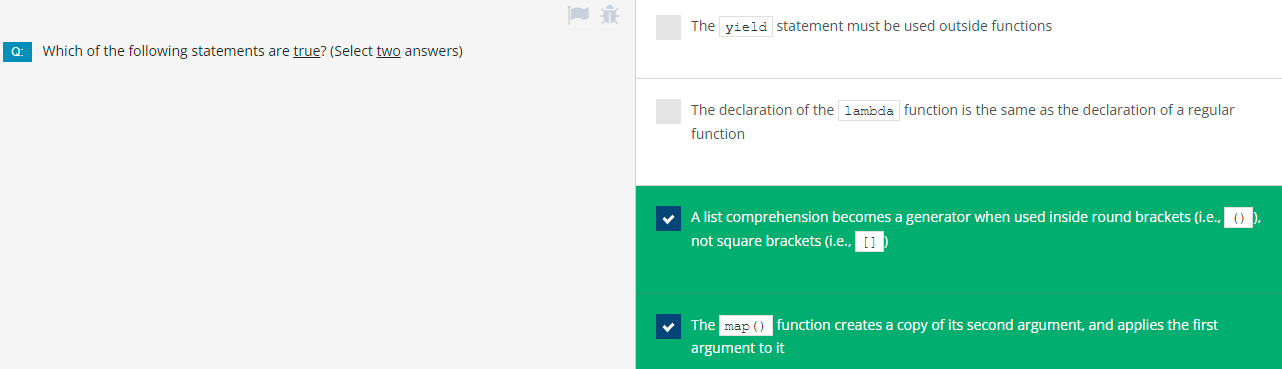


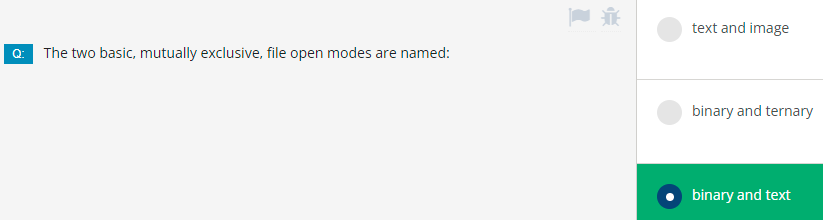


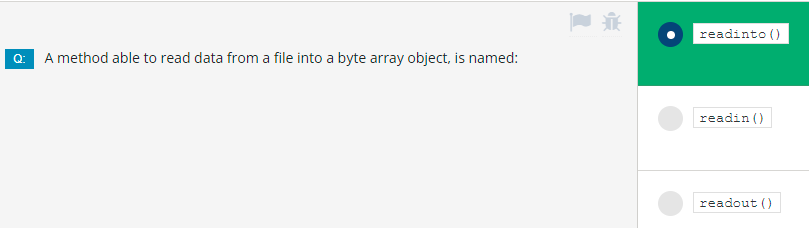


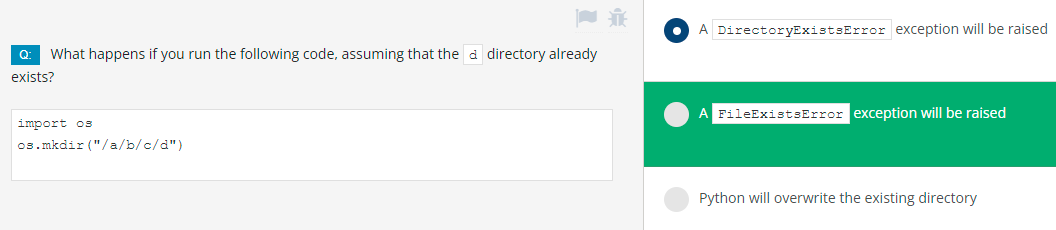


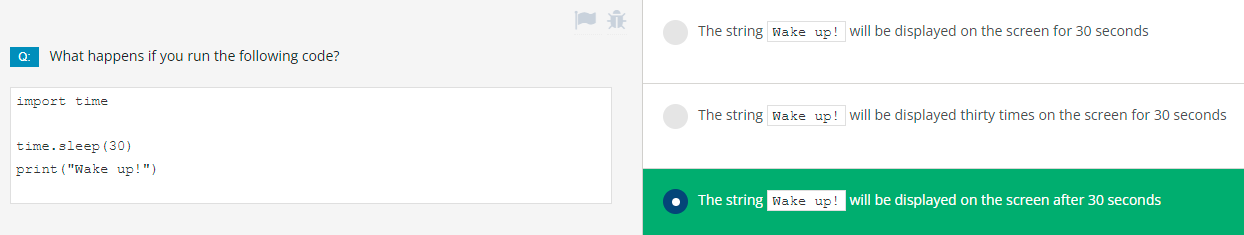


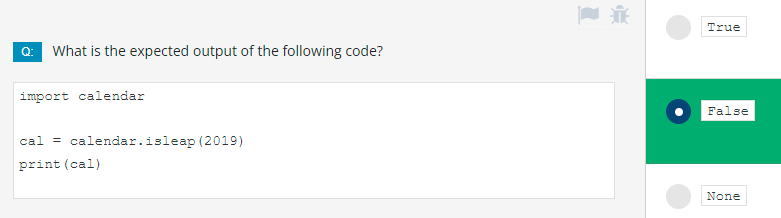












# Module 4 Test

PE2 各章统计

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 |  | 3 |  | 4 |  |
| 2.1 | **4** | 3.1 | 8 | 4.1 | 15 |
| 2.2 | **15** | 3.2 | 16 | 4.2 | 12 |
| 2.3 | **18** | 3.3 | 9 | 4.3 | 18 |
| 2.4 | **6** | 3.4 | 15 | 4.4 | 9 |
| 2.5 | **11** | 3.5 | 23 | 4.5 | 23 |
| 2.6 | **12** | 3.6 | 10 | 4.6 | 15 |
| 2.7 | **8** |  |  |  |  |
| 2.8 | 6 |  |  |  |  |
|  | 80 |  | 81 |  | 92 |