Chapter 1 - Modules and Packages

In this chapter, I will help you summarize all contents of this module 1. You will review six main content, including importing variants, advanced qualifying for nested modules; dir(), the sys.path variable; some math functions as ceil(), floor(), trunc(), factorial(), hypot(), sqrt(); some random functions as random(), seed(), choice(), sample(); some platform functions as platform(), machine(), processor(), system(), version(), python_implementation(), python_version_tuple(); rationale (why do we need modules?), __pycache__, __name__, public variables, __init__.py and finally, searching for/through modules/packages; nested packages vs directory tree. Your main task is to read carefully again and complete quiz module 1 alone to check your understanding and memory.

Remember that this exam module is the first section of the exam. It constitutes a maximum of 12% of the overall exam score. It contains six (06) single-choice and multiple-choice items, each graded 1 or 2 points.

Specifications

1. Section weight: 12%
2. Maximum raw score: 12
3. Questions drawn: 6
4. Question types: single-choice and multiple-choice
5. Points per question: 1 + 1 (multiple-choice), 2 (single-choice and multiple-choice)

Let's start your journey now!

A. The essentials of modules and packages

6. 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 no two objects with the same name).

7. A module is a file containing Python definitions and statements which can be imported and used by another Python code.

8. A package is a way of structuring a module's namespace by using dotted module names. It can be said that the module is a hierarchical collection of modules.

9. To import a module and to make its content available, the import directive is used. Let's assume that there is a module named mod which contains a function named fun(). The following import forms can be used:

- **import mod**
  - The clause contains the import keyword followed by the name of the imported module (or a comma-separate list of names);
  - When any of the imported module's entities (variables, functions, classes, etc.) should be used, its name must be encoded in the dotted (qualified) form, e.g. mod.fun()
- **from mod import fun**
  - The clause starts with the phrase from module_name followed by the name (or a comma-separated list of names) of the imported module's entity/entities;
  - When any of the imported names is used inside the code, its name must be used as is, without any dots or qualification, e.g. fun();
  - Note: unless you know all the names provided by the module, you may not be able to avoid name conflicts inside your current namespace.
- **from mod import ***
  - The clause is similar to the previously presented variant, but uses an asterisk (*) instead of any explicit name/names. The asterisk works as a wild card and implicitly imports all the module's entities;
  - The names of any of the imported entities must be used as is - no qualification is allowed.

10. A built-in function named dir() can be used to obtain an alphabetically sorted list which contains all entity names available in the module passed to the function as an argument, e.g. print(dir(match)) will print a list of the math module's contents.

B. The math module

The math module comes with the standard Python release, and contains some useful constants and functions for carrying out mathematical operations. Note: all trigonometrical functions take their arguments expressed in radians.

- math.pi $->$ pi constant value
- math.e $->$ Euler's number value.
- math.sqrt(x) $->$ the square root of x
- math.sin(x) $->$ the sine of x
- math.cos(x) $->$ the cosine of x
- math.tan(x) $->$ the tangent of x
- math.asin(x) $->$ the arcsine of x
- math.acos(x) x$->$ the arccosine of x
- math.atan(x) $->$ the arctangent of x
- math.radians(x) $->$ a function that converts x from degrees to radians
- math.degrees(x) $->$ acting in the other direction (from radians to degrees)
- math.sinh(x) $->$ the hyperbolic sine
- math.cosh(x) $->$ the hyperbolic cosine
- math.tanh(x) $->$ the hyperbolic tangent
- math.asinh(x) $->$ the hyperbolic arccosine
- math.atanh(x) $->$ the hyperbolic arctangent
- math.exp(x) $->$ finds the value of e^x
- math.log(x) $->$ the natural logarithm of x
- math.log(x, b) $->$ the logarithm of x to base b

- math.log10(x) −> the decimal logarithm of x (more precise than math.log(x, 10))
- log2(x) −> the binary logarithm of x (more precise than math.log(x, 2))
- math.ceil(x) −> the ceiling of x (the smallest integer greater than or equal to x)
- math.floor(x) −> the floor of x (the largest integer less than or equal to x)
- math.trunc(x) −> the value of x truncated to an integer (be careful - it's not an equivalent of either ceil or floor)
- math.factorial(x) −> returns x! (x has to be an integral and not a negative)
- math.hypot(x) −> returns the length of the hypotenuse of a right-angle triangle with the leg lengths equal to x and y (the same as math.sqrt(pow(x, 2) + pow(y, 2)) but more precise)

C. The random module

1. The random module delivers some mechanisms allowing you to operate with pseudorandom numbers.
2. 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 the chosen algorithms) and produces a new seed value. The length of a cycle in which all seed values are unique may be very long, but it is finite - sooner or later the seed values will start repeating, and the generating values will repeat, too. The initial seed value, set during the start of the program, determines the order in which the generated values will appear.
3. The seed() function directly sets the generator's seed. Possible variants of its invocation are:
   - seed() - sets the seed with the current time which makes the seed a bit unpredictable;
   - seed(int_value) −> sets the seed with the integer value int_value
4. The random module's pseudorandom generator is available through the following functions:
   - random() −> produces a float number x, which falls within the range $0.0 \leqslant x < 1.0$

Example

```
>>> import random

>>> random.seed(0)
```

>>> print(random.random())

The code will always print the same value, regardless of the number of launches.

- random.randrange(start, stop, step) − > produces an integer number x, which is taken from the range $start \leqslant x < stop$ with step step. Note: the start argument defaults to 0 and the step argument defaults to 1

Example: all of the invocations below are equivalent:

>>> import random

>>> print(random.random(100))

>>> print(random.random(0, 100))

>>> print(random.random(0, 100, 1))

- random.randint(start, stop) − > equivalent of random.randrange(start, stop + 1);
- choice(sequence) − > chooses a "random" element from the input sequence (e.g. a list or tuple) and returns it;
- sample(sequence, elements_to_choose = 1) − > returns a list (a sample) consisting of the elements_to_choose elements (which defaults to 1) "drawn" from the input sequence.

D. The platform module

The platform module lets you access the underlying platform's data, that is, the hardware, the operating system, and the interpreter version information. Some of the module's functions are:

- platform.platform(aliased = False, terse = False) − > returns a string describing the underlying hardware architecture and OS.
  - platform.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).
- platform.machine() − > returns a string with the genetic name of the host processor.
- platform.processor() − > returns a string with the real name of the host processor.

- platform.system() − > returns a string with the version of the host OS.
- platform.python_implementation() − > returns a string denoting the Python implementation (expect Python here, unless you decide to use any non-canonical Python branch).
- platform.python_version_tuple() − > returns a three-element tuple filled with:
    - The major part of Python's version;
    - The minor part;
    - The patch level number.

The complete list of currently available, community-led Python modules can be found here.

E. How Python deals with modules

1. When a certain module is imported for the first time, Python convert its contents into a semi-compiled code which can be used to execute module functions faster.
2. The semi-compiled files are placed in the directory named __pycache__, located in the same directory in which the source module exists.
3. If the source file of a module is named mod.py, its semi-compiled counterpart will be named mod.cpython-xy.pyc where x and y are numbers derived from your Python's version number.
4. There is a special built-in variable named __name__ whose value:
    - is set to "__main__" when the module is run as a standalone code;
    - is set to the source file's name (excluding .py) when the file is imported as a module.

Example: the snippet below can be used to determine the context in which the source file is used

```
>>> if __name__ == "__main__":
>>>     print("Run as a program")
>>> else:
>>>     print("Imported as a module")
```

5. If the name of any module entities starts with _ or __, the entity should be treated as private and not used outside the module.

6. The very first line of a Python source file can start with #!, in which case it's called the shabang, shebang, hashbang, poundbang, or even the hashpling line. From Python's point of view, it's just a comment. For Unix and Unix-like OSs (including MacOS) such a line instructs the OS on how to execute the contents of the file (in other words, what program needs to be launched to interpret the text). This is what it may look like in some environments:

```
>>> #! /usr/bin/env python3
```

7. The module of the name given in the import directive is searched inside directories whose names are listed in the sys.path variable (it's a list strings). The first element of the list is the directory in which the code that performs the import resides. The user is allowed to modify the sys.path contents in order to limit or extend the range of directories being searched through.

F. Packages

1. A group of Python source files deployed in a certain subtree of the file system may form a package. Let's assume that the following set of files and directories exists:

```
[any directory]
    └── package          ← directory
        ├── gearbox.py   ← file: contains turn_on() function
        ├── __init__.py  ← empty file
        └── subpackage   ← directory
            └── engine.py ← file: contains start() function
```

2. A source file named __init__.py located in certain place of the directory structure defines the root of the package, in other words, it determines the location and name of the package. If the file is not empty, its contents will be executed when any of the package's modules is imported, so it may be used to initialize the package's state.

3. Importing a module from within a package requires more detailed import specifications. If we assume that the package directory is listed in the sys.path variable, it's guaranteed that it will be searched through when Python is looking for modules.

- If you want to invoke the turn_on() function provided by the gearbox.py module located inside the package, this is the way you can do it:

```
>>> import package.gearbox

>>> package.gearbox.turn_on()
```

• An alternative way of achieving almost the same effect is the following:

>>> from package.gearnox import turn_on

>>> turn_on()

Note: the difference in the invocation syntax.

• This form of import will work, too:

>>> from package.gearbox import *

>>> start()

• Invoking the start() function delivered by the engine.py module provided by the subpackage subpackage existing in the package can be gained in the following three ways:

Example 1:

>>> import package.subpackage.engine

>>> package.subpackage.engine.start()

Example 2:

>>> from package.subpackage,engine import start

>>> start()

Example 3:

>>> from package.subpackage.engine import *

>>> start()

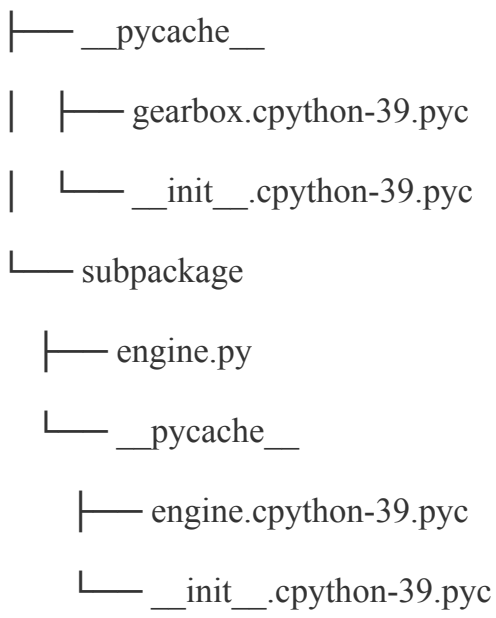
4. Executing the imports above (Example 1 - 3) will provoke Python to create two __pycache__ directories located under the package and sub package directories, and will fill them with .pyc files, which contain semi-compiled code of the modules imported, In effect, the subtree contents structure will look like this:

[any directory]

└── package

├── gearbox.py

├── __init__.py


__pycache__
gearbox.cpython-39.pyc
__init__.cpython-39.pyc
subpackage
engine.py
__pycache__
engine.cpython-39.pyc
__init__.cpython-39.pyc

Quiz Chapter 1

Question 1: A function named f() is included in a module named m, and the module is a part of a package called p. Which of the following code snippets allows you to invoke the function properly? (Select two answers)

A. from p.m import f

   f()

B. import p.m.f

   f()

C. import p

   m.f()

D. import p.m

   p.m.f()

Question 2: The following code snippet is used to import and invoke the c() function. What is always true about the function and its environment? (Select two answers)

```
>>> import a.b
>>> a.b.c()
```

A. The c() function is contained in the b module.

B. The c() function can also be invoked in the following way: c()

C. The b component is the module, and it is contained inside the package or is a submodule of a

D. The c() function can also be invoked in the following way: b.c()

Question 3: What is TRUE about the built-in dir() mechanism in the context of modules and packages?

A. It is a method that can be invoked from within a module to obtain the module contents.

B. It is a function that can be invoked with a module passed as an argument to obtain the module contained.

C. It is a dictionary contained by a module reflecting the module contents.

D. It is a list contained by a module reflecting the module contents.

Question 4: What is true about sys.path?

A. It is a string that contains the name of the directory/folder where the Python installation is deployed.

B. It is a string that copies the OS PATH environment variable.

C. It lists strings that indicate all directories/folders scanned by Python to find the specific modules.

D. It is a string that contains the name of the OS Current Working Directory/ Folder.

Question 5: Which of the following functions come from the math module? (Select two answers)

A. processor()

B. seed()

C. sqrt()

D. hypot()

Question 6: A programmer needs to use the following functions: machine(), choice(), and system(). Which modules have to be imported to make this possible? (Select two answers)

A. tkinter

B. platform

C. math

D. random

Question 7: What is the expected output of the following code?

```
>>> import math
>>> x = -1.5
>>> print(abs(math.floor(x) + math.ceil(x)))
```

A. 3

B. 2

C. -3

D. -2

Question 8: What is the expected output of the following code?

```
>>> import random
>>> random.seed(0)
>>> x = random.choice([1,2])
>>> random.seed(0)
>>> y = random.choice([1,2])
>>> print(x-y)
```

A. -1

B. 1

C. 0

D. True

Question 9: Which of the following statements is TRUE? (Select two answers)

A. A programmer must manually create a directory/folder named __pycache__ inside every package.

B. The variable named __name__ is a string containing the module name.

C. The .pby extension marks files that contain Python semi-compiled byte-code.

D. A source file named __init__.py is used to mark a directory/folder containing a Python package and initialize the package.

Question 10: Which statements are TRUE about the __pycache__ directory/folder? (Select two answers)

A. It contains semi-compiled module code.

B. It has to be created manually by the module's user.

C. It has to be created manually by the module's creator.

D. It is created automatically.

Question 11: What is true about how Python looks for modules/packages?

A. The directory from which the code has been run is searched only upon the user's request.

B. The directory from which the code has been run is always searched through.

C. The directory from which the code has been run is never searched through.

D. The directory from which the code has been run must not contain any modules.

Question 12: Which of the following variables will Python consider to be **private**?

A. _privatedata_

B. private__data

C. __privatedata

D. privatedata__

Chapter 2 - Exceptions

The chapter's objectives will cover all of the exceptions with the following principal contents: except, finally, except: except, except - else, except(e_1, e_2); the hierarchy of exceptions; raise, raise ex, assert; event classes, except E as e, the arg property; self-defined exceptions; defining and using user-created exceptions. As in chapter 1, your main task is to read carefully again and complete quiz module 2 alone to check your understanding and memory.

This exam chapter is the second section that appears in the exam. It constitutes a maximum of 14% of the overall exam score. It contains five (5) single-choice and multiple-choice items, and each one can be graded 1, 2, or 4 points.

Specifications

1. Section weight: 14%
2. Max raw score: 14
3. Questions drawn: 5
4. Question types: single choice and multiple choice
5. Points per question: 1 + 1 (multiple choice), 2 (single choice), 2 + 2 (multiple choice), or 4 (single choice).

A. Exceptions

6. In Python, exceptions are events which break the normal course of code execution.

7. Exceptions can be handled using the try-except clause or left alone, which causes abnormal program termination.

8. To ensure that an exception is handled in a controlled way, the following syntax has to be used (the parts of the code highlighted are optional):

```
>>> try:
>>>     #The part of the code which can raise an exception.
>>> except:
>>>     #The part of the code designed to handle a possible exception.
>>> else:
>>>     #The part of the code executed when no exception is raised.
>>> finally:
>>>     #The part of the code that is *always* executed.
```

4. The else: and finally: branches are not obligatory and can be omitted.

5. Code example:

```
>>> x, y = 1, 0
>>> try:
>>>     result = x / y
>>> except:
>>>     print("ZeroDivisionError")
>>> else:
>>>     print("Success")
>>> finally:
>>>     print(“Completed")
```

- The sample code prints ZeroDivisionError when the y variable is zero, and Success otherwise. The text Completed is always printed, no matter whether the exception has been raised or not.

- The text Success is presented when the y variable is non-zero.

- Note that the finally: branch, if it exists, is executed even when any of the other branches causes an error and raises an exception.

6. All Python exceptions are dedicated class objects, and the classes form a hierarchy.

7. The top-most class is named BaseException, and some of its direct subclasses are:

- Exception, which gathers events caused by program execution failures;

- KeyboardInterrupt, which is raised when the user triggers an OS action designed to interrupt a running program (e.g., presses the Ctrl-C key combination).

- SystemExit is raised implicitly when a Python program executes the exit() function.

8. Some of the most useful Exception class subclasses are:

9. ArithmeticError, which may appear during mathematical evaluations – its subclasses describe more specific failures like:

- ZeroDivisionError, which should be self-explanatory;

- OverflowError is raised when the result of any operation is too large to be stored in the computer memory (e.g., 2. ** 100000).

10. LookupError, which is connected to problems caused by improper use of data aggregates. It includes:

- IndexError, which is raised when the code tries to access a non-existent sequence element, for example, if lst is an empty list, the expression lst[0] will raise IndexError;

- KeyError, which is when the situation applies to non-sequential collections like dictionaries. For example, if dir is an empty directory, the expression dir["key"] will raise KeyError.

11. TypeError, which is triggered by operations which misuse argument types (e.g. "1" > 1)

12. ImportError, which is caused by invalid import directives (e.g., import nonexistent module).

13. AssertionError, which is raised when the assert instruction fails (e.g., assert True == False).

14. AttributeError, which appears when the program performs an invalid attribute reference (e.g., object.nonexistent_attribute = 0).

15. FileNotFoundError is raised when you try to open a non-existent file using "r" mode.

16. When more than one exception may be raised inside one try: block, and each of the exceptions requires a different treatment, another variant of try-except is used (the parts of the code highlighted are optional):

```
>>> try:
>>>     #The part of the code which can raise more than one exception.
>>> except Exception_1:
>>>     #The part of the code designed to handle Exception_1 exceptions.
>>> except Exception_n
>>>     #The part of the code designed to handle Exception_n exceptions.
>>> except:
>>>     #The default part of the code is designed to handle other (unhandled) exceptions.
>>> else:
>>>     #The part of the code executed when no exception is raised.
>>> finally:
>>>     #The part of the code that is *always* executed.
```

17. No other branch will be executed when an exception goes to one of the enlisted except branches.

18. When there is no except Exception_n branch, which matches the raised exception, and there is no default except branch, the exception remains unhandled and triggers the default Python behavior unless it is handled in a further part of the code.

19. Code example:

```
>>> x, y = 1, 0
>>> try:
>>>     value = int(input("Enter value: "))
```

```
>>>     print("Reciprocal of the value is", 1 / value)

>>> except ValueError:

>>>     print("ValueError")

>>> except ZeroDivisionError:

>>>     print("ZeroDivisionError")

>>> except:

>>>     print("An unidentified exception")

>>> else:

>>>     print("Success")

>>> finally:

>>>     print("Completed")
```

20. The sample code prints ValueError when the user inputs a string that cannot be converted into an integer value.

21. If the user enters 0, the code prints ZeroDivisionError.

22. If any other exception is raised (e.g., the user presses Ctrl-C instead of entering a value) and the default except branch is present, the text An unidentified exception is printed. The message that reads Completed is printed unconditionally.

23. When the user inputs a reasonable integer value and the reciprocal is successfully calculated, the result is outputted to the screen, and if the else: branch exists, the text Success followed by Completed appears on the screen.

24. The else: branch (if it exists) must be specified after all the except branches.

25. The default except branch must be specified after all the except Exception_n: branches.

26. The finally branch (if it exists) must be specified as the last of all branches.

Remember: the order in which the beaches are listed matters!

27. As Python analyzes exception branches in the same order in which they appear in the source code, it means than more general except Exception_n branches should follow the most specialized ones, in other words, knowing that ZeroDivisionError is a subclass of

ArithmeticError, it is recommended to put the latter branch before the former one, like this:

```
>>> try:
>>>     pass
>>> except ZeroDivisionError:
>>>     pass
>>> except ArithmeticError:
>>>     pass
```

28. Swapping the branches will make the second branch unreachable, like here:

```
>>> try:
>>>     pass
>>> except ArithmeticError:
>>>     pass
>>> except ZeroDivisionError:
>>>     pass
```

29. Note that ZeroDivisionError will go to the ArithmeticError branch in this case, as Python chooses the first matching branch and does not search for any (possibly) better match.

30. If you want to handle more than one exception in the same except branch, you can use the following syntax:

```
>>> try:
>>>     #Some code.
>>> except (Exception_1, Exception_2):
>>>     #Handling two different exceptions occur here.
```

31. Code example:

```
>>> try:
```

```
>>>     value = float(input("Enter value: "))
>>>     print(2. / value)
>>>     print(2. ** value)
>>> except ZeroDivisionError:
>>>     print("more specialized")
>>> except ArithmeticError:
>>>     print("more general")
>>> except:
>>>     print("other")
>>> else:
>>>     print("success")
```

- The sample code prints more specialized when the user inputs 0.
- The text more general is outputted when the user enters values large enough to evoke the OverflowError exception during a power calculation. The exception is a subclass of the ArithmeticError class to go to the second exception branch.
- Placing ArithmeticError in the first branch will no longer allow the code to enter the ZeroDivisionError branch.
- The text other will be printed when the exception raised does not match any of the previous branches.
- The text success appears when the calculation is successful, and no exception has been raised.

Note: Exceptions are collected together in a way which resembles a tuple.

32. The assert instruction is used to check necessary conditions for further program execution. In addition, it is used to protect selected parts of the code from data that can lead to erroneous code behavior. The instruction's syntax is:

```
>>> assert condition
```

33. If the condition evaluates to a value that is neither zero, None, nor an empty string, the program continues its execution.

34. If the condition is anything else, the execution is suspended, and the AssertionError exception is raised. The exception can be handled routinely, just like any other one.

#Example

```
>>> try:
>>>     value = float(input("Enter value: "))
>>>     assert value != 0  # can be shortened to assert value
>>>     print("Reciprocal of the value is", 1 / value)
>>> except AssertionError:
>>>     print("Invalid value")
```

35. The sample code prints an Invalid value when the user inputs 0 and prints the reciprocal calculation result otherwise.

36. Any built-in or user-defined exceptions can be raised manually on demand using the raise instruction. The instruction's syntax is:

```
>>> raise exception
```

37. The instruction can be used in any context, including except branches.

#Example

```
>>> try:
>>>     value = float(input("Enter value: "))
>>>     if value == 0:
>>>         raise ZeroDivisionError
>>>     print("Reciprocal of the value is", 1 / value)
>>> except ZeroDivisionError:
>>>     print("Invalid input")
```

38. The sample code prints Invalid input when the user inputs 0. Note that the exception has been raised manually before the division.

39. The specialized, argumentless variant of the raise instruction can be used only during ongoing exception handling. In other words, it is illegal to execute it outside except

branches. The instruction is designed to re-raise the same exception currently being handled. It allows the programmer to handle the exception partially and explicitly delegate further handling to other parts of the code.

40. As exceptions can propagate through function boundaries, any exception may be handled outside the function in which it was raised.

Note: try-except blocks can be nested. which means that certain blocks can be placed in any part of other blocks.

41. Code example:

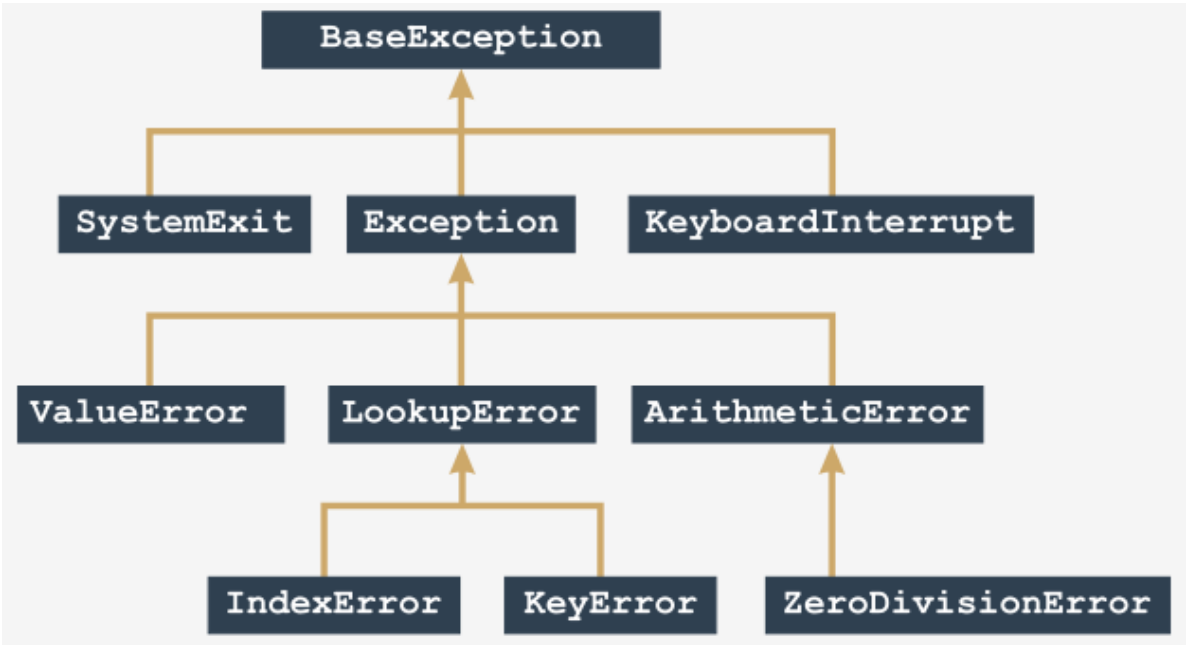
```
>>> def top():
>>>     try:
>>>         raise ZeroDivisionError
>>>     except:
>>>         print("at the top")
>>>         raise
>>> def middle():
>>>     top()
>>> try:
>>>     top()
>>> except:
>>>     print("on the bottom")
```

The sample code prints the following lines:

```
>>> at the top
>>> on the bottom
```

- The first output line is printed inside the top() function while handling the manually raised ZeroDivisionError exception.
- The main part of the ode produces the second output line as the re-raised exception leaves the top() function, oversteps the middle() function body, and is finally handled by the except branch at the lowest level of code execution.

42. In general, exceptions are objects of built-in class hierarchy, which form a tree with the BaseException class as the top-most class (root class). For example, a part of the tree that presents some of the most common exceptions is shown below:



43. An extended except syntax exists that is dedicated to handling an exception and catching the exception's object, which carries additional information allowing you to explore its details. It looks like this:

```
>>> try:
>>>     #The part of the code which can raise an exception.
>>> except ExceptionClass as exception_object:
>>>     #The part of the code designed to handle the possibly raised exception.
```

44. The exception_object variable is an instance of the ExceptionClass class, with all the consequences of this fact.

45. The __str__() of the exception classes is defined in a way that lets you explore the informative message associated with a particular specific exception.

46. Code example:

```
>>> s = "test string"
>>> try:
>>>     s = s[100]
```

```
>>> except Exception as exc:
>>>     print(exc)
```

The sample code will print string index out of range.

47. Some usable exception class properties are:

- the __name__ variable, which corresponds to the exception's name;
- the __subclasses__() function, which returns a list of objects representing all the exception's subclasses;
- moreover, instances of exception classes contain the args variable, a tuple that gathers all arguments passed to the exception class constructor (self is not counted here, so if the constructor receives no arguments, the tuple is empty).

48. Code example:

```
>>> print(ArithmeticError)
>>> print(ArithmeticError.__subclasses__())
>>> try:
>>>     raise ZeroDivisionError
>>> except Exception as exc:
>>>     print(exc.args)
>>> try:
>>>     raise ZeroDivisionError(2, "nd exception")
>>> except Exception as exc:
>>>     print(exc.args)
```

The code produces the following output:

```
>>> <class 'ArithmeticError'>
>>> [<class 'FloatingPointError'>, <class 'OverflowError'>, <class 'ZeroDivisionError'>]
>>> ()
>>> (2, 'nd exception’)
```

The output contains:

- a reference to the ArithmeticError class;
- a list of references to the ArithmeticError sub-classes;
- an empty args tuple from the object created by default with an argumentless constructor invocation;
- a two-element args tuple from the object created with a two-argument constructor invocation.

49. Users can extend the existing exception hierarchy to cover their specific needs. Any existing exception classes can be used as a base for the new (sub)tree. However, it makes sense to choose only these exceptions related to the programming problem domain (e.g., it's a bad idea to use ZeroDivisionError as a base for exceptions connected to network communication).

50. If the new hierarchy should not be closely connected to Python's exception tree, it is reasonable to derive it from any of the top-level classes, like Exception.

51. The newly created class exceptions follow the routine rules governing the classification of exceptions.

52. Code example:

```
>>> class RangeError(IndexError):
>>>     pass


>>> class Collection:
>>>     def get(self, index):
>>>         if not (1 <= index <= 10):
>>>             raise RangeError
>>>         return 42


>>> stuff = Collection()
>>> try:
```

```
>>>     print(stuff.get(1))

>>>     print(stuff.get(0))

>>> except IndexError as error:

>>>     print(“failure")
```

The code produces the following output:

```
>>> 42

>>> failure
```

The output contains:

- a value returned by the normal .get() invocation;
- a message produced by the IndexError exception handler (note that the exception handled is actually an IndexError subclass).

53. It is also possible to equip the newly defined exception with a richer set of attributes than the original when required by the problem domain.

54. Code example:

```
>>> class RangeError(IndexError):

>>>     __errors = 0

>>>     def __init__(self, the_index):

>>>         IndexError.__init__(self, "erroneous index: " + str(the_index))

>>>         RangeError.__errors += 1

>>>     def get_error_counter(self):

>>>         return RangeError.__errors


>>> class Collection:

>>>     def get(self, index):

>>>         if not (1 <= index <= 10):
```

```
>>>             raise RangeError(index)

>>>         return 42

>>> stuff = Collection()

>>> try:

>>>     print(stuff.get(1))

>>>     print(stuff.get(0))

>>> except RangeError as error:

>>>     print("failure")

>>>     print(error)

>>>     print(error.get_error_counter())
```

The code produces the following output:

```
>>> 42

>>> failure

>>> erroneous index: 0

>>> 1
```

The output contains:

- a value returned by the normal .get() invocation;
- a message produced by the RangeError exception handler;
- a string representation of the exception object produced by the superclass's __str__() function invocation;
- a value returned by the .get_error_counter() function.

Quiz Chapter 2

Question 1: Which of the following operations may raise an exception? (Select two answers)

A. Indexing a list.

B. Slicing a string.

C. Invoking the int() function.

D. Incrementing an integer variable by one.

Question 2: Which of the following are the names of built-in Python exceptions? (Select two answers)

A. LookupException.

B. KeyError.

C. AssertionError.

D. ProgramTooComplicatedError.

Question 3: What is the expected output of the following code?

```
>>> x, y = 3.0, 0.0
>>> try:
>>>     z = x/y
>>> except ArithmeticError:
>>>     z = 1
>>> else:
>>>     z = -2
>>> print(z)
```

A. -1

B. An error message appears on the screen

C. -2

D. +INF

Question 4: What is the expected output of the following code?

```
>>> x, y = 0.0, 3.0
>>> try:
>>>     z = x/y
>>> except ArithmeticError:
>>>     z = -1
>>> else:
>>>     z = -2
>>> print(z)
```

A. +INF

B. An error message appears on the screen

C. -2

D. -1

Question 5: What is the expected output of the following code?

```
>>> def fun(x):
>>>     return 1/x
>>> def mid_level(x):
>>>     try:
>>>         fun(x)
>>>     except:
>>>         raise AssertionError
>>>     else:
>>>         return 0
```

```
>>> try:
>>>     x = mid_level(0)
>>> except Exception:
>>>     x = -1
>>> except:
>>>     x = -2
>>> print(x)
```

A. 0

B. -2

C. -1

D. An error message appears on the screen

Question 6: What is the excepted output of the following code?

```
>>> def fun(x):
>>>     assert x >= 0
>>>     return x ** 0.5
>>> def mid_level(x):
>>>     try:
>>>         fun(x)
>>>     except Error:
>>>         raise
>>> try:
>>>     x = mid_level(1)
>>> except RuntimeError:
>>>     x = -1
```

```
>>> except:
>>>     x = -2
>>> print(x)
```

A. -1

B. An error message appears on the screen

C. 0

D. -2

Question 7: What is the expected output of the following code?

```
>>> consts = (3.141592, 2.718282)
>>> try:
>>>     print(consts[2])
>>> except Exception as exception:
>>>     print (exception.args)
>>> else:
>>>     print("('success')")
```

A. ('success')

B. ('tuple index out of range',)

C. 2.718282

D. 3.141592

Question 8: What is the expected output of the following code?

```
>>> consts = [3.141592, 2.718282]
>>> try:
>>>     print(consts.index(314e-2))
>>> except Exception as exception:
```

```
>>>     print(exception.args)
>>> else:
>>>     print("success")
```

A. ('3.14 is not in list',)

B. -1

C. False

D. ('success')

Question 9: Which of the following messages will appear on the screen when the code is run? (Select two answers)

```
>>> class Accident (Exception):
>>>     def __init__(self, message):
>>>         self.message = message
>>>     def __str__(self):
>>>         return "problem"
>>> try:
>>>     print("action")
>>>     raise Accident ("accident")
>>> except Accident as accident:
>>>     print(accident)
>>> else:
>>>     print("success")
```

A. accident

B. success

C. problem

D. action

Question 10: Which of the following messages will appear on the screen when the code is run? (Select two answers)

```
>>> class Failure(IndexError):
>>>     def __init__(self, message):
>>>         self.message = message
>>>     def __str__(self):
>>>         return “problem”
>>> try:
>>>     print(“launch”)
>>>     raise Failure(“ignition”)
>>> except RuntimeError as error:
>>>     print(error)
>>> except IndexError as error:
>>>     print(“ignore”)
>>> else:
>>>     print(“landing”)
```

A. ignore

B. problem

C. ignition

D. launch