Chapter 3 - Strings

Objectives covered by this chapter will help you review some main contents about the string methods as ASCII, UNICODE, UTF-8, codepoints, and escape sequences; ord(), chr(), and literals; indexing, slicing, and immutability; iterating through; concatenating, multiplying, comparing (against strings and numbers); in, not in; .isxxx(), .join(), .split(), .sort(), sorted(), .index(), .find(), and .rfind(). As the above chapter, you must have read carefully again and complete quiz module 3 alone to check your understanding and memory.

This exam chapter is the second section that appears in the exam. It constitutes a maximum of 18% of the overall exam score. It contains eight (8) single-choice and multiple-choice items, and each one can be graded 1, 2, or 4 points.

Specifications

1. Section weight: 18%

2. Max raw score: 18

3. Questions drawn: 8

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).

Strings

1. Characters (alphabetical signs, digits, punctuation marks, emojis, etc.) are stored in the computer memory as numbers. The way in which certain numbers are assigned to corresponding characters is called an encoding standard. A number assigned to a certain character according to a given encoding standard is a code point.

2. In the distant past, it was common thing for hardware manufacturers to use their different proprietary encoding standards. For example, IBM developed a standard named Extended Binary-Coded Decimal Interchange Code (EBCDIC) which is still in use today on IBM mainframes and their clones.

3. A commonly adopted and globally used encoding standard is American Standard Code for Information Interchange (ASCII) whose modern version uses eight bits for its code points and is able to define 256 different characters. A complete list of ASCII code points can be found here: https://en.wikipedia.org/wiki/ASCII.

4. There are some interesting facts about ASCII which are worth mentioning:

   • decimal digit code points occupy values from 48 (0x30) ("0") to 57 (0x39) ("9")

   • the space code point is 32 (0x20)

   • upper-case Latin letters occupy the code points from the range 65 (0x41) ("A") to 90 (0x5A) ("Z").

   • lower-case Latin letters use values from 97 (0x61) ("a") to 122 (0x7A) ("z").

   • note that any lower-case letter code point is greater by 32 (a space code point) than its upper-case counterpart.

5. As ASCII is not able to cover the majority of symbols which are in use in languages other than English (especially those which use alphabets different than Latin) a new, more general encoding standard has been established – its name is Unicode. ASCII is a Unicode subset. Any ASCII code point has the same meaning in Unicode, but not vice versa. Unicode covers more than 143,000 characters and is accepted by nearly all currently used hardware and software products.

6. A physical representation of Unicode characters is defined outside Unicode itself and is subject to additional standards known as Unicode Transformation Standards (UTF). Depending on the number of bits used to carry code points, a few transformations have been defined, such as UTF-32, UTF-16, and UTF-8 (note that the latter is the dominant encoding on the WWW).

7. Python 3 accepts UTF-8 encoded source files. All named Python entities like variables, functions, classes, etc. can contain letters derived from non-Latin alphabets, although it has to be mentioned that such practice may break PEP-8 rules.

8. Python string literals can be:

- single-line when the literal fits entirely inside one line of the source code;
- multi-line when the literal may span over more than one line of the source code.

9. Single-line literals are delimited either by a pair of apostrophes ('string') or by a pair of quotes ("string"). Note that PEP-8 makes no special recommendation about the preferred way of quoting strings – pick a rule yourself and stick to it!

10. Multi-line literals are delimited either by a pair of triple apostrophes ('''string''') or by a pair of triple quotes (“""string""").

11. Apostrophe-delimited strings can contain quotes and vice versa.

12. An empty string is a string that contains no characters. All the strings below are empty:

```
>>> ''

>>> ""

>>> ''''''

>>> """"""
```

13. If an apostrophe has to be embedded inside an apostrophe-delimited string, the inner apostrophe must be preceded (escaped) by a backslash ('King\'s College'). The same rule applies to quotes located inside quote-delimited strings.

14. The backslash is also used to embed control characters (characters whose ASCII code points are less than 32) inside strings. Some of the characters are:

- "\n" (newline – NL) – produces a line break on terminals and inside text files;
- "\r" (carriage return – CR) – moves the cursor back to the beginning of the line;
- "\t" (horizontal tabulation – HT) – moves the cursor to the next tab stop;
- "\f" (form feed – FF) – loads a new page of papers on printers;
- "\a" (bell – BEL) – makes a beep on terminals.

15. To be properly interpreted, the backslash itself has to be doubled when put inside a string (“\\").

16. If a multi-line string spans more than one source code line, it contains an adequate number of newline characters ("\n").

17. A couple of functions are used to transform code points into characters and vice versa:

- ord(c) which returns a character's code point as an integer value;
- chr(i) which returns a single character string obtained from a given code point.

18. There are two equalities that are always true:

- >>> ord(chr(i)) == i
- >>> chr(ord(c)) == c

19. The length of a string is determined by the len() function:

- len("") evaluates to 0;
- len("string") evaluates to 5.

20. Python strings are sequences – this means that they can be subject to:

- indexing;
- slicing;
- iterating through.

21. Any of the characters contained in the string can be extracted by string indexing:

- "string"[1] evaluates to "t";
- "string"[-2] evaluates to “n".

Using the wrong index (an index which goes beyond string boundaries) causes an IndexError exception.

22. Any string can be sliced. The slicing produces a new string, leaving the original string untouched:

- "string"[2:4] evaluates to "ri";
- "string"[::2] evaluates to "srn";
- "string"[::-1] evaluates to "gnirts".

23. Note: the slice which breaks string boundaries does not raise an exception – the non-existent characters are replaced by empty strings instead:

- "string"[2:100] evaluates to "ring";

- "string"[-100:-200] evaluates to an empty string.

24. The for loop can be used to traverse (iterate) the string.

#Example

```
>>> sample = "string"

>>> for char in sample:

>>>     print(char, end="_")
```

The above code prints: s_t_r_i_n_g_.

25. Python strings are immutable. This means that all of the following snippets will raise an exception:

```
>>> sample = "string"

>>> sample[0] = "S" # TypeError
```

```
>>> sample = "string"

>>> del sample[0] # TypeError
```

```
>>> sample = "string"

>>> sample.append("s") # AttributeError
```

26. Data other than a string can be converted into strings or their string representation can be obtained using the str() function:

#Example

```
>>> import math

>>> print(len(str(math.pi)))
```

This code prints 17 on the screen.

27. The in and not in operators can be applied to strings – the left operand is a string (substring) which is searched for, and the right is a string which is searched through.

#Example 1

```
>>> "s" in "string"        # True

>>> "S" not in "string"    # True

>>> "ring" not in "string" # False

#Example 2

>>> multi_line = """

>>> """

>>> print("\n" in multi_line)
```

The above code (Example 2) prints True to the screen.

28. It can be said that an empty string is contained inside every string, even an empty one. Note that the expression below is always true, even if the right operand is an empty string:

```
>>> "" in string
```

29. Note that c not in s is not the equivalent of not c in s – be aware of this!

30. There are two string operators in Python:

- +, which performs string concatenation:

```
>>> string + string → string
```

- *, which performs string multiplication:

```
>>> int * string → string

>>> string * int → string
```

Code example:

```
>>> string_1 = "string"

>>> string_2 = "STRING"


>>> print(string_2 + string_1)

>>> print(2 * string_1)
>>> print(string_2 * 2)

>>> print(2 * string_1 + 0 * string_2)
```

The sample code produces the following output:

```
>>> STRINGstring

>>> stringstring

>>> STRINGSTRING

>>> stringstring
```

Note: The two string operators can be used in shortcut form: += and *=

```
#Example 1

>>> my_string = "I♡U"

>>> my_string += "2"

>>> print(my_string)  # Outputs: I♡U2

#Example 2

>>> my_string = "I♡U"

>>> my_string *= 2

>>> print(my_string)  # Outputs: I♡UI♡U
```

31. Some of the functions which can be applied to strings are:

- min(str) – finds the minimal character (in a code points value sense) of the string (the string cannot be empty!)
- min('012ABCxyz') evaluates to '0';
- max(str) – behaves like min() but finds the maximal character of the argument;
- max('012ABCxyz') evaluates to 'z';
- list(str) – creates a list, and each element of the list contains one character of the argument string;

- list('abc') evaluates to ['a', 'b', ‘c'].

String Methods

Some of the methods which can be invoked from within strings are:

- string.index(str) returns the index of the first occurrence of the str argument inside the string object, or raises a ValueError exception if the argument cannot be found;
- Example: "012ABCxyz".index("AB") evaluates to 3.
- string.count(str) counts all occurrences of the strargument inside the string object;
- Example: "alpha beta gamma".count("a") evaluates to 5.
- string.capitalize() returns a copy of the string object with its first character capitalized and the rest lower-case;
- Example: "tHe".capitalize() evaluates to "The".
- string.center(size) returns a copy of the string object centered inside a field of size width; Example: "Heading".center(15) evaluates to "   Heading   ".
- string.center(size, char) performs the same task as the previous variant, but fills the empty space with char instead of space;
- Example: "Heading".center(15,"-") evaluates to "----Heading----".
- string.endswith(str) checks if a given string object ends with the specified str substring, and returns True or False, depending on the check result;
- Example: "the end of the world".endswith("collapse") evaluates to False.
- string.startswith(str) checks if a given string starts with the specified str substring;
- Example: "the end of the world".startswith("the") evaluates to True.
- The result of string.find(str) is the same as that returned from the .index() method, but evaluates to -1 when the substring is not found instead of raising a ValueError exception;
- Example: "Where are the snows of yesteryear?".find("ar") evaluates to 6.
- string.rfind(str) does the same search as .find(), but starts its activity from the end of the string;
- Example: "Where are the snows of yesteryear?".rfind("ar") evaluates to 32.

The string methods that begin with is

Methods whose names start with is are designed to check if a string meets certain characteristics, and return True or False depending on the check result:

- string.isalnum() checks if a string contains only digits or alphabetical characters (letters);
- Example: "R2D2".isalnum() evaluates to True.
- string.isalpha() works like .isalnum(), but is interested in letters only;
- Example: "R2D2".isalpha() evaluates to False.
- string.isdigit() looks for decimal digits only;
- Example: "42".isdigit() evaluates to True.
- string.islower() is like .isalpha(), but accepts lower-case letters only;
- Example: "islower".islower() evaluates to True.
- string.isupper() is like .islower(), but its objects of interests are upper-case letters;
- Example: "Upper?".isupper() evaluates to False.
- string.isspace() identifies whitespaces only;
- Example: " \t\n\r".isspace() evaluates to True.

A group of string methods creates a new string object using the original string contents as a base for different transformations:

- string.lower() returns a copy of the string with all the upper-case characters converted to lower-case;
- Example: "The End".lower() evaluates to "the end".
- string.upper() returns a copy of the string with all the lower-case characters converted to upper-case;
- Example: "The End".upper() evaluates to "THE END".
- string.swapcase() makes a new string by swapping the case of all letters within the source string;
- Example: "tHE eND".swapcase() evaluates to "The End".

- string.title() returns a title-cased version of the string;
- Example: "the end".title() evaluates to "The End".
- string.lstrip() returns a copy of the string with leading spaces removed;
- Example: " The End ".lstrip() evaluates to "The End ".
- string.rstrip() returns a copy of the string with trailing spaces removed;
- Example: " The End ".rstrip() evaluates to " The End".
- string.strip() combines the effects of both .rstrip() and .lstrip();
- Example: " The End ".strip() evaluates to "The End".
- string.replace(str1, str2) returns a copy of the original string in which all occurrences of the str1 argument have been replaced by the str2 argument;
- Example: "Big Bang".replace("B", "D") evaluates to "Dig Dang".

Two string methods require special attention:

- string.join(list) returns a string which is the concatenation of the strings contained in the list. If there are no strings in the list, a TypeError exception is raised. The separator between the joined elements of the result is the string itself.
- Example: ",".join(["Little", "John"]) evaluates to "Little,John"
- string.split(sep) returns a list of the words found in the string, using sep as the delimiter string; if the sep argument is ommitted, a space is used as the default delimiter;
- Example: "Little,John".split(",") evaluates to ["Little", "John"].
- Example: "Little,John".split() evaluates to [“Little,John"].

A complete reference of Python string methods can be found here.

Quiz Chapter 3

Question 1: Which of the following are character encoding standard names? (Select two answers)

A. ASCII

B. Unicode

C. Intcod

D. Unilang

Question 2: Which of the following are **TRUE** about string encoding used by Python? (Select two answers)

A. The Python escape sequence always begins with the \ character.

B. ASCII and Unicode code points have nothing character in common.

C. Codepoint is a number which corresponds to a character.

D. UTF-8 encoding cannot be used in Python source files.

Question 3: Which of the following expressions evaluate to **TRUE**? (Select two answers)

A. chr(ord('k') + 2) == 'i'

B. ord('x') - ord('x') == ord('')

C. ord('x') + ord('x') == ord('')

D. chr(ord('k') + 2) == 'm'

Question 4: Which of the following are **valid** Python string literals? (Select two answers)

A. """The Knights Who Say 'Ni!'"""

B. "\"

C. "King's Cross Station"

D. 'All the king's horses'

Question 5: Which of the following assignments can be performed without raising any exceptions? (Select two answers)

A. s = 'rhyme'

s[0] = s[1]

B. s = 'rhyme'

s[9]

C. s = 'rhyme'

s = s[::2]

D. s = 'rhyme'

s = s[-2]

Question 6: Which of the following snippets output ABC to the screen? (Select two answers)

A. print ('ABCDEF'[3:])

B. print ('FEDCBA'[-3:]

C. print ('AXBYCZ'[::-2]

D. print ('AXBYCZ'[::2]

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

```
>>> plane = "Cessna"

>>> counter = 0

>>> for c in plane * 2:

>>>     if c in ["e", "a"]:

>>>         counter += 1

>>> print(counter)
```

A. The code is erroneous and cannot be run

B. 0

C. 2

D. 4

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

```
>>> plane = "Blackbird"

>>> counter = 0

>>> for c in plane + 2:

>>>     if c in ["1", "2"]:

>>>         counter += 1

>>> print(counter)
```

A. 2

B. 0

C. The code is erroneous and cannot be run

D. 4

Question 9: Which of the following expressions evaluates to **FALSE** and raises no exception?

A. 'All' * 2 <= 'Alan'

B. '9' * 1 <= 1 * 2

C. 10 == '1' + '0'

D. '9' * 3 < '9' * 9

Question 10: Which of the following expressions evaluate to **TRUE** and raise no exception? (Select two answers)

A. str(None) == None

B. 'Analog' < 'analog'

C. str(None) != 'None'

D. '' * 0 < 1 * ''

Question 11: Which of the following expressions evaluate to **TRUE** and raise no exception? (Select two answers)

A. '' not in ''

B. 'bc' in 'abc'

C. 'xyz' not in 'uvwxyz'

D. ' ' in 'alphabet'

Question 12: Which of the following expressions evaluate to **FALSE** and raise no expression? (Select two answers)

A. not ('a' not in 'abc')

B. '\n' in """

"""

C. '123' in '1-2-3'

D. not 'a' in 'abc'

Question 13: What is the expected output of the following code?

```
>>> foo = "Mary had 21 little sheep"

>>> print(foo.split()[2].isdigist())
```

A. True

B. False

C. 2

D. 21

Question 14: What is the expected output of the following code?

```
>>> strng = '\'.join(("Mary", "had", "21", "sheep"))

>>> print(strng)[0:1}.islower())
```

A. False

B. True

C. M

D. Mary

Question 15: Which of the following can be used to find a given substring within a string? (Select two answers)

A. The .index() method

B. String slicing

C. The .rfind() method

D. The find() function

Question 16: Which of the following snippets can be used to build a new string consisting of sorted characters contained in the 'xyz' string assigned to the letters variable? (Select two answers)

>>> letters = 'xyz'

A. tmp = list(letters)

   tmp.sort()

   new_string = ''.join(tmp)

B. new_string = sorted(letters)

C. new_string = ''.join(sorted(letters))

D. tmp = letters.sort()

   new_string = str(tmp)

Chapter 4 - Object - Oriented Programming

In this chapter, I will cover ll of the OOP with the following principal contents: ideas about class, object, property, method, encapsulation, inheritance, grammar vs class, superclass, subclass; instance vs class variables as declaring, initializing; methods like declaring, using the self parameter; introspection: hasatr() (objects vs classes), __name__, __module__, __bases__properties; inheritance: single, multiple, isinstance(), overriding, not is and is operators; constructors: declaring and invoking; polymorphism; the __str__() method; multiple inheritance, diamonds.

This exam chapter is the fourth section that appears in the exam. It constitutes a maximum of 34% of the overall exam score. It contains twelve (12) single-choice and multiple-choice items, each one can be graded 1, 2, or 4 points.

Specifications

1. Section weight: 34%
2. Max raw score: 34
3. Questions are drawn: 12
4. Question types: single- and multiple-choice
5. Points per question: 1+1 (multiple-choice), 2 (single-choice), 2+2 (multiple-choice), or 4 (single-choice)

Object - Oriented Programming (OOP)

1. Object-Oriented Programming (OOP) is a programming paradigm (a distinct set of coding techniques) based on the concept of objects. An object is an entity which combines data and code.

   The data is stored in the form of fields (also known as attributes or properties), and code is represented in the form of functions (also known as methods).

   The bundling of data and the methods which operate on that data, or restricting direct access to the data, is called encapsulation.

```
# Example

>>> class TheClass:

>>>     class_variable = True  # This is a class variable (property).

>>> def __init__(self):

>>>     self.instance_variable = False  # This is an instance variable.

>>> def do_this(self):  # This is a class function (method).

>>>     return TheClass.class_variable
```

2. An object is an incarnation of ideas expressed in a class, while a class is a kind of recipe which can be used to create an object.

```
# Example

>>> class TheClass:

>>>     pass

>>> the_object = TheClass()  # the_object is an object of TheClass class.
```

3. Classes form a hierarchy in which the more specialized classes (known as subclasses) are placed below the more general or abstract classes (known as superclasses). Note: the hierarchy grows from top to bottom, like the roots of a tree, rather than the branches.

```
# Example

>>> class A:

>>>     class_variable = 1
>>>     def do(self):
>>>         self.instance_variable = 2;
>>> class B(A):
>>>     pass
>>> o = B()
>>> o.do()
>>> print(o.class_variable, o.instance_variable)
```

4. Inheritance is a mechanism of basing a class upon another class, retaining a similar implementation and a common set of traits. The process can be also defined as deriving new classes (subclasses) from existing ones (superclasses or base classes).

5. In the inheritance hierarchy, a class's component that is defined later (in the inheritance sense) overrides the same component that has been defined earlier.

6. Single inheritance happens when a certain class has only one direct superclass.

7. Multiple inheritance takes place when a class has more than one direct superclass.

8. The diamond problem term describes issues that may derive from multiple inheritance. The problem appears when a class is a subclass of more than one superclass derived from one common super-super-class.

9. Polymorphism is a mechanism which enables the programmer to modify the behavior of any of the object's superclasses without modifying these classes themselves.

10. Composition is the process of composing an object using other different objects.

11. Introspection is the ability of a program to examine the type or properties of an object at runtime.

12. Reflection is the ability of a program to manipulate the values, properties, and/or functions of an object at runtime.

13. Every existing object may be equipped with three groups of attributes:

    - a name that uniquely identifies it within its home namespace;

    - a set (maybe empty) of individual properties which make it original and unique;

• a set of methods to perform specific activities, able to change the object itself, or some of the other objects.

14. The simplest Python class can be defined in the following way:

```
>>> class TheClass:

>>>     pass
```

15. The instantiation (creation of the object which is an instance of the class) is done in the following way:

```
>>> the_object = TheClass()
```

16. Any class or object component (either a method or property) must be accessed using dotted notation (i.e. the_object.component).

17. Python objects, when created, are gifted with a small set of predefined properties and methods. 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.

18. Each class (not object!) is equipped with a string variable named __name__, which contains the name of the class itself, and with a tuple named __bases__, which contains classes (not class names!) which are direct superclasses of the class.

19. All classes and objects have a string variable named __module__, which stores the name of the module containing the definition of the class.

20. An instance variable is a kind of property which is closely connected to the object (class instance), not to the class itself. Each of the objects (even when the objects are derived from the same class) can have a different set of instance variables. Modifying the instance variable of any object has no impact on all the remaining objects.

21. A class variable is a property which exists in just one copy and is stored outside any object. Class variables aren't shown in an object's __dict__ dictionary – they can be found inside the class's __dict__ dictionary. Remember that a class variable always presents the same value in all class instances (objects).

22. A variable (either a class or instance) becomes private (inaccessible directly from outside the class/object) if its name starts with two underscores (__). Such a variable can be unhidden by the mechanism called name mangling. For example, a property named __property defined inside an object named the_object, being an instance of the TheClass class, can be accessed directly as:

```
>>> the_object._TheClass__property
```

Mangling won't work if you add an instance variable outside the class code.

23. Referencing to a non-existent class/object component raises the AttributeError exception. To check a component's existence, the hasattr() function can be used, which expects two arguments to be passed to it:

    - the class or the object being checked;
    - the string with the name of the property whose existence has to be reported.

The function returns either True or False.

24. A method is a function embedded inside a class. Every Python method is obliged to have at least one parameter – a method may be invoked without an argument, but must not be declared without parameters). The first (or even only) parameter is usually named self, and the name suggests the parameter's purpose – it identifies the object for which the method is invoked.
25. 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.

26. The self parameter is used to obtain access to the object's instance and class variables, as well as to invoke other object/class methods from inside the class.
27. A method called __init__() is a constructor. If a class has a constructor, it is invoked automatically and implicitly when the object of the class is instantiated.
28. When a class has a superclass and:

    - it doesn't have its own constructor, then a superclass constructor is invoked implicitly during class creation;
    - has its own constructor, then a superclass constructor has to be invoked explicitly.

29. The constructor:

    - is obliged to have the self parameter (it's set automatically, as usual)
    - may 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, in other words, to properly initialize its internal state, create instance variables, instantiate any other objects if their existence is needed, etc.

- cannot return a value;
- cannot be invoked directly either from the object or from inside the class – you can only invoke a constructor from any of the object's superclass constructors.

30. When Python needs any class/object to be presented as a string, it tries to invoke a method named __str__() from the object and to use the string it returns. The method can be overridden to construct adequate and useful behavior.

```
# Example

>>> class A:

>>>     pass

>>> class B:

>>>     def __str__(self):

>>>         return "object!"

>>> a = A()

>>> b = B()

>>> print(a, b)
```

The code above outputs <__main__.A object at 0x7ffa07c3adc0> object! to the screen.

31. To define a class named SubClass as a subclass of a superclass or superclasses, the following syntax has to be used:

```
# Example 1

>>> class Subclass(SuperClass):

>>>     pass
```

Or

```
# Example 2

>>> class Subclass2(SuperClass_1, Superclass_2):

>>>     pass
```

32. To check if a particular class is a subclass of any other class, a function named issubclass(ClassOne, ClassTwo) is used. The function returns True if the ClassOne class is a subclass of the ClassTwo class, and False otherwise.

```
# Example

>>> class A:

>>>     pass

>>> class B(A):

>>>     pass

>>> print(issubclass(A, B), issubclass(B, A), issubclass(B, B))
```

Note: each class is considered to be a subclass of itself.

33. To detect if an object is an instance of a certain class, a function named isinstance(the_object, TheClass) is used. The function returns True if the_object object is an instance of the TheClasclass, or False otherwise.

```
# Example

>>> class A:

>>>     pass

>>> class B:

>>>     pass

>>> o = A()

>>> print(isinstance(o, A), isinstance(o, B))
```

The code above outputs True False to the screen.

34. The `is` operator checks whether two variables refer to the same object. If they do, the expression evaluates to `True`, and to `False` otherwise.

```
# Example

>>> class A:

>>>     pass
>>> o1 = A()

>>> o2 = o1

>>> o3 = A()

>>> print(o1 is o2, o1 is o3)
```

The code above outputs True False to the screen.

35. The super() function, when used inside any class methods, gives access to the superclass without needing to know its name. It can be said that the super() function creates a context in which you don't have to pass the self argument to the method being invoked – this is why it's possible to activate the superclass constructor using only one argument.

    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.

```
# Example

>>> class A:

>>>     def __init__(self):

>>>         print("A's __init__()")

>>> class B(A):

>>>     def __init__(self):

>>>         super().__init__()

>>>         A.__init__(self)

>>>         pass

>>> o = B()
```

The code presented above outputs A's __init__() twice to the screen.

36. The Method Resolution Order (MRO for short) is a strategy in which a particular programming language scans through the upper parts of a class's hierarchy in order to find the method it currently needs. Python does this in the following order:

- find it inside the object itself;
- find it in all classes involved in the object's inheritance line from bottom to top.

(If there is more than one class inside a certain level of the inheritance, the classes at this level are scanned from left to right in the order in which they have been put in the inheritance declaration.)

If both of the above fail, an exception (AttributeError) is raised.

```
# Example

>>> class A:

>>>     def do(self):

>>>         print("A")

>>> class BL(A):

>>>     def do(self):

>>>         print("BL")

>>> class BR(A):

>>>     def do(self):

>>>         print("BR")

>>> class C(BR, BL):

>>>     pass

>>> o = C()

>>> o.do()
```

The code above outputs BR to the screen.

37. When the class tries to impose an order different than that described above, a TypeError is raised with a message which reads:

```
>>> Cannot create a consistent method resolution order (MRO) for bases xxx and yyy.
```

Note: Due to an illegal C class inheritance declaration, the code below causes TypeError to be raised:

```
# Example

>>> class A:
```

```
>>>     pass
>>> class B(A):
>>>     pass
>>> class C(A, B):
>>>     pass
>>> o = C()
```

Replacing class C(A, B) with class C(B, A) removes the error.

Quiz Chapter 4

Question 1: What is TRUE about object-oriented programming (OOP)? (Select two answers)

A. A class is like a blueprint used to construct objects.

B. A relation between a superclass and its subclass is know as fraternity.

C. A class may exist without its objects, while objects cannot exist without their class.

D. Polymorphism is a phenomenon which allows you to have many classes of the same name.

Question 2: What is TRUE about object-oriented programming (OOP)? (Select two answers)

A. A part of a class designed to build new objects is called a constructor.

B. A superclass of a class is located below the class in the hierarchy diagram.

C. Encapsulation is a phenomenon which allows you to hide certain class traits from the outer world.

D. All objects of the same class have exactly the same set of attributes.

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

```
>>> class Tin:
>>>     label = "soup"
>>>     def __init__(self, prefix):
>>>         self.name = prefix + " " + Tin.label


>>> can_1 = Tin("Tomato")
>>> can_2 = Tin("Chicken")
>>> print(can_1.label == can_2.label)
```

A. It outputs False

B. The code is erroneous and it will raise an exception

C. It outputs None

D. It outputs True

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

```
>>> class Package:
>>>     spam = ''
>>>     def __init__(self, content):
>>>         Package.spam += content + ";"
>>> envelope_1 = Package("Capacitors")
>>> envelope_2 = Package("Transistors")
>>> print(envelope_2.spam)
```

A. It outputs Capacitors;

B. It outputs Capacitors; Transistors;

C. The code is erroneous and it will raise an exception

D. It outputs Transistors;

Question 5: Assuming that the following code has been executed successfully, indicate the expressions which evaluate to True and don't raise any exceptions. (Select two answers)

```
>>> class Collection:
>>>     stamps = 2
>>>     def __init__(self, stuff):
>>>         self.stuff = stuff
>>>     def dispose(self):
>>>         del self.stuff
>>> binder = Collection(1)
>>> binder.dispose()
```

A. len(binder.__dict__) > 0

B. len(binder.__dict__) != len(Collection.__dict__)

C. ‘stamps’ in Collection.__dict__

D. ‘Stuff’ in binder.__dict__

Question 6: Assuming that the following code has been executed successfully, indicate the expressions which evaluate to TRUE and don’t raise any exceptions. (Select two answers).

```
>>> class Class:
>>>     class_var = 1
>>>     def __init__(self):
>>>         self.instance_var = 1
>>>     def method(self):
>>>        pass
>>> object = Class()
```

A. len(object.__dict__) == len(Class.__dict__)

B. Class.__dict__[‘method’] != None

C. ‘__dict__’ in Class.__dict__

D. object.__dict__[‘method’] != None

Question 7: Given the code below, indicate the code lines which correctly increment the __element variable by one. (Select two answers)

```
>>> class BluePrint:
>>>     __element = 1
>>>     def __init__(self):
>>>         self.component = 1
>>>     def __action(self):
>>>        pass
>>> product = BluePrint()
```

A. product._BluePrint__element += 1

B. _product__element += 1

C. BluePrint._BluePrint__element += 1

D. BluePrint.element += 1

Question 8: Given the code below, indicate the code line which correctly invokes the __action() method.

```
>>> class BluePrint:
>>>     __element = 1
>>>     def __init__(self):
>>>         self.component = 1
>>>     def __action(self):
>>>        pass
>>> product = BluePrint()
```

A. BluePrint._BluePrint__action()

B. product._BluePrint__action()

C. product._product__action()

D. product.__action()

Question 9: Given the class below, indicate a method which will correctly provide the value of the rack field?

```
>>> class Storage:
>>>     def __init__(self):
>>>         self.rack = 1
>>>     #Insert a method here.
>>> stuff = Storage()
>>> print(stuff.get())
```

A. def get(self):

```
return self.rack
```

B. def get():

```
return rack
```

C. def get():

```
return self.rack
```

D. def get(self):

```
return rack
```

Question 10: Given the class below complete the print() method body in a way that will ensure that the get() method is properly invoked. (Select two answers)

```
>>> class Storage:
>>>     def __init__(self):
>>>         self.rack = 1
>>>     def get(self):
>>>         return self.rack
>>>     def print(self):
>>>         #Insert a line of code here.
>>> stuff = Storage()
>>> print(stuff.print())
```

A. print(Storage.get())

B. print(self.get())

C. print(Storage.get(self))

D. print(get())

Question 11: The built-in class properly called __base__ is:

A. A variable of type int, which stores the radix currently used by the class

B. A tuple, which contains information about the direct superclasses of the class.

C. A string, which contains information about the direct superclass of the class.

D. A dictionary, which contains information about the all superclasses of the classes

Question 12: What is the expected output of the following code?

```
>>> class Ceil:
>>>     Token = 1
>>>     def get_token(self):
>>>         return 1
>>> class Floor(Ceil):
>>>     def det_token(self):
>>>         return 2
>>>     def set_token(self):
>>>         pass
>>> holder = Floor()
>>> print(hasattr(holder, "Token"), hasattr(Ceil, “set_token"))
```

A. True True

B. True False.

C. False True

D. False False

Question 13: What is the expected output of the following code?

```
>>> class Economy:
>>>     def __init__(self):
>>>         self.econ_attr = True
>>> class Business(Economy):
```

```
>>>     def __init__(self):
>>>         super().__init__()
>>>         self.busn_attr = False
>>> econ_a = Economy()
>>> econ_b = Economy()
>>> busn_a = Business()
>>> busn_b = busn_a
>>> print(isinstance(busn_a, Economy) and isinstance(econ_a, Business), end = " ")
>>> print(busn_b is busn_a or econ_a is econ_b)
```

A. False True

B. True True

C. False False

D. True False

Question 14: Given the code below, which of the expressions will evaluate to TRUE? (Select two answers)

```
>>> class Top:
>>>     value = 3
>>>     def say(self):
>>>         return self.value
>>> class Middle(Top):
>>> value = 2
>>> class Bottom(Middle):
>>>     def say(self):
>>>         return -self.value
>>> short = Bottom()

>>> tall = Top()

>>> average = Middle()
```

A. short.value == 2

B. isinstance(average, Bottom)

C. tall.say() == 2

D. average.value == 2

Question 15: Given the code below, which of the following expressions will evaluate to TRUE? (Select two answers)

```
>>> class Alpha:

>>>     value = "Alpha"

>>>     def say(self):

>>>         return self.value.lower()

>>> class Beta(Alpha):

>>> value = "Beta"

>>> class Gamma(Alpha):

>>>     def say(self):

>>>         return self.value.upper()

>>> class Delta(Gamma, Beta):

>>> pass

>>> d = Delta()

>>> b = Beta()
```

A. isinstance(d, Beta)

B. Alpha in Delta.__bases__

C. d.say() == “BETA”

D. d.value == “Alpha”

Question 16: Given the code below, which of the following expressions will evaluate to TRUE? (Select two answers)

```
>>> class Alpha:

>>>     def say(self):

>>>         return "alpha"

>>> class Beta(Alpha):

>>>     def say(self):

>>>         return "beta"

>>> class Gamma(Alpha):

>>>     def say(self):

>>>         return "gamma"

>>> class Delta(Beta, Gamma):

>>> pass

>>> d = Delta()

>>> b = Beta()
```

A. b is d

[✓ B.] Gamma in Delta.__bases__

C. d.say() == “gamma”

[✓ D.] isinstance(d, Alpha)

Question 17: Which of the following classes have valid constructors? (Select two answers)

A. class Dalet:

```
    def__init__(self):

      return False
```

B. class Bet:

def__init__(self):

return ArithmeticError

C. class Gimel:

def__init__():

self.attribute = True

D. class Aleph:

def__init__(self):

self.attribute = True

Question 18: Which of the following snippets output TRUE to the screen? (Select two answers)

A. class A:

def __init__(self, value = True):

print(value)

a = A()

B. class A:

def __init__():

print(True)

a = A()

C. class A:

def __init__(self, value = True):

print(value)

class B(A):

def __init__(self, value = False):

supper().__init__(value)

b = B()

```
D.  class A:
        def __init__(self):
            print(True)
    class B(A):
        pass
    b = B()
```

Question 19: What is the expected behavior of the following snippet?

```
>>> class Team:
>>>     def show_ID(self):
>>>         print(self.get_ID())
>>>     def get_ID(self):
>>>         return "anonymous"
>>> class A(Team):
>>>     def get_ID(self):
>>>         return "Alpha"
>>> a = A()
>>> a.show_ID()
```

A. It outputs Alpha

B. It outputs anonymous

C. It raises an exception

D. It outputs an empty line

Question 20: What is the expected output of the following code?

```
>>> class Aircraft:
>>>     def start(self):
```

```
>>>         return "default"

>>>     def take_off(self):

>>>         start()

>>> class Fixed_Wing(Aircraft):

>>> pass

>>> class Rotor_Craft(Aircraft):

>>>     def start(self):

>>>         return "spin"

>>> fleet = [Fixed_Wing(), Rotor_Craft()]

>>> for airship in fleet:

>>>     print(airship.start(), end = " “)
```

A. spin default

B. spin spin

[✓ C]. default spin

D. default default

Question 21: If you want an object to be able to present its contents as a string, you should equip its class with a method named:

A. str()

B. __tostring__()

[✓ C]. __str__()

D. string()

Question 22: If you want to check if a Python file is either used as a module or run as a standalone program, you should check a built-in variable named:

[✓ A]. __name__

B. __run_mode__

C. __used_as__

D. __module_name__

Question 23: Given the class hierarchy presented below, indicate which of the following class declarations are legal. (Select two answers)

```
>>> class Top:

>>>     pass

>>> class Left(Top):

>>>     pass

>>> class Right(Top):

>>>     pass
```

A. class Bottom(Left, Right):

        pass

B. class Bottom(Top, Right):

        pass

C. class Bottom(Top, Left):

        pass

D. class Bottom(Left, Top):

        pass

Question 24: What is the expected output of the following code?

```
>>> class Top:

>>>     def __str__(self):

>>>         return '1'

>>> class Left(Top):

>>>     def __str__(self):

>>>         return '2'
>>> class Right(Top):
>>>     def _str__(self):
>>>         return '3'
>>> class Bottom(Right, Left):
>>>     pass
>>> object = Bottom()
>>> print(object)
```

A. 1

B. 2

C. An empty line

D. 3

Chapter 5 - Miscellaneous

Objectives covered by this chapter will help you review some main contents about list comprehensions: the if operator, using list comprehensions; lambdas: defining and using lambdas, self-defined functions taking lambda as arguments; map(), filter(); closures: meaning; defining, and using closures; I/O Operations: I/O modes, predefined streams, handles; text/binary modes; open(), errno and its values, close(); .read(), .write(), .readline(), readlines(), and bytearray().

This exam block is the fifth and the last section that appears in the exam. It constitutes a maximum of 22% of the overall exam score. It contains nine (9) single-choice and multiple-choice items, each one can be graded 1, 2 or 4 points.

Specifications

- Section weight: 22%
- Max raw score: 22
- Questions are drawn: 9
- Question types: single- and multiple-choice
- Points per question: 1+1 (multiple-choice), 2 (single-choice), 2+2 (multiple-choice), or 4 (single-choice)

Generators

1. A Python generator is a class containing a piece of specialized code able to produce a series of values, and to control the iteration process. The generator returns a series of values, and in general, is (implicitly) invoked more than once.

2. An iterator protocol is a way in which an object should behave in order 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.

3. 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.

4. For example, the following code implements an iterator that generates the desired number of first Fibonacci sequence elements.

```
>>> class Fib:
>>>   def __init__(self, last):
>>>       self.__last = last
>>>       self.__current = 0
>>>       self.__prev_1 = self.__prev_2 = 1
>>>   def __iter__(self):
>>>       return self
>>>   def __next__(self):
>>>       self.__current += 1
>>>       if self.__current > self.__last:
>>>           raise StopIteration
>>>       if self.__current in [1, 2]:
```

```
>>>         return 1
>>>     return_value = self.__prev_1 + self.__prev_2
>>>     self.__prev_1, self.__prev_2 = self.__prev_2, return_value
>>>     return return_value
>>> for i in Fib(10):
>>>     print(i, end=" “)
```

The code produces the following output:

```
>>> 1 1 2 3 5 8 13 21 34 55
```

The yield statement

1. The main disadvantage of the iterator protocol is the need to save the state of the iteration between subsequent __iter__() invocations. This is why Python offers a much more effective, convenient, and elegant way of writing iterators. The concept is based on a 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 – a function containing the yield statement should not be invoked explicitly as, in fact, it isn't a function anymore; it's a generator object.

2. The invocation of such a function will return the object's identifier, not the series we expect from the generator.

3. Values returned by subsequent generator invocations can be turned into a list using the list() function.

4. Example: this is what a very simple generator may look like:

```
>>> def double_step(n):
>>>     for i in range(0, 2*n, 2):
>>>         yield i
>>> for v in double_step(5):
>>>     print(v, end=" “)
```

The code prints the desired number of values, which start from 0 and form an arithmetic progression with a difference of 2.

5. Example: a Fibonacci series generator which utilizes the yield concept may be implemented like this:

```
>>> ddef Fib(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
>>> print(list(Fib(10)))
```

Its output is exactly the same as the one produced by the previous variant of the code.

List comprehensions

1. List comprehension is an element of Python syntax which allows the programmer to create a new list either using the values of an existing list or generated "on the fly" by the for clause of another generator expression.

   Example: the following comprehension:

   ```
   >>> new_list = [10 ** ex for ex in range(3)]
   ```

   produces the following new list: [1, 10, 100], and assigns it to the new_list variable.

   The effects of this comprehension expression are equivalent to these produced by the code snippet shown below:

   ```
   >>> new_list = []
   >>> for element in range(3):
   >>>     new_list.append(10 ** element)
   ```

2. To control the process of comprehension, the if clause can be used.

Example: the following comprehension:

```
>>> new_list = [10 ** ex for ex in range(3) if ex % 2 != 0]
```

appends a new element to the newly created list only when x is an odd value. Its equivalent looks like this:

```
>>> new_list = []

>>> for element in range(3):

>>>     if element % 2 != 0:

>>>         new_list.append(10 ** element)
```

Both variants set the new_list variable which contains only one element: [10]

3. The if clause used outside the comprehension presents more flexible behavior, and may take the following form:

   expression_1 if condition else expression_2 which is in fact an operator evaluating to expression_1 when the condition is true, and expression_2 otherwise.

   Example:

   ```
   >>> new_list = []

   >>> for x in range(10):

   >>>     new_list.append(1 if x % 2 == 0 else 0)
   ```

   sets the new_list variable with the following list:

   ```
   >>> [1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
   ```

The lambda function

1. In Python, a lambda function is a function without a name (an anonymous function). Its declaration takes the following form:

   ```
   >>> lambda parameters : expression
   ```

   Such a clause returns the value of the expression when taking into account the current values of the lambda's arguments. Note that the parameters part can be empty – in this case, the lambda function takes no arguments at all.

   Example:

```
>>> just_two  = lambda : 2

>>> square_of = lambda x : x * x

>>> power_of  = lambda x, y : x ** y

>>> for a in range(3):

>>>     print(square_of(a), power_of(a, just_two()))
```

The output of the snippet looks like this:

```
>>> 0 0

>>> 1 1

>>> 4 4
```

2. A lambda function can be passed to another function as an argument, and invoked. Such a technique may simplify the code, and allows the programmer to avoid writing functions which are invoked only once.

```
>>> def print_x_y(arg, function):

>>>     print("f(" + str(arg) + ")=" + str(function(arg)))

>>> print_x_y(3, lambda x: 1 / x)

>>> print_x_y(3, lambda x: x * x)

>>> print_x_y(3, lambda x: x ** x)
```

The above snippet produces the following output:

```
>>> f(3)=0.3333333333333333

>>> f(3)=9

>>> f(3)=27
```

The map() function

1. The map() function applies a function (e.g. a lambda) passed by its first argument to all the elements of its second argument, 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.

Example: The code presented below:

```
>>> import math

>>> list_x = [x for x in range(5)]

>>> list_y = list(map(lambda x: x * x, list_x))

>>> print(list_y)

>>> for x in map(lambda x: int(math.sqrt(x)), list_y):

>>>     print(x, end=' ')

>>> print()
```

prints the following two lines to the screen:

```
>>> [0, 1, 4, 9, 16]

>>> 0 1 2 3 4
```

The sample shows the power of lambdas – the same algorithm, encoded without lambdas, would be longer and less legible.

The filter() function

1. The filter() function, as the name suggests, filters its second argument (e.g. a list) while being guided by directions flowing from the function (e.g. a lambda) specified as the first argument (the function is invoked for each list element, just like in map()). The list's elements for which the function returns True pass the filter, and the other elements are rejected.

   The filter() function returns an iterator – just like the map() function.

   Example:

   ```
   >>> list_in = [x for x in range(-5, 6, 2)]

   >>> list_out = list(filter(lambda x: x > 0, list_in))

   >>> print(list_out)
   ```

   The above snippet's output looks like this:

   ```
   >>> [1, 3, 5]
   ```

   The filter() function accepts only those elements whose values are greater than 0, and the other elements are rejected.

Closures

1. 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.

2. Example:

```
>>> def outer(parameter):
>>>     local_var = parameter
>>>     def inner():
>>>         return local_var
>>>     return inner
>>> function = outer(1)
>>> print(function())
```

   Explanation:

   - the inner() function returns the value of the local_var variable, accessible inside its scope, as inner() can use any of the entities at the disposal of the outer() function;
   - the outer() function returns a copy of the inner() function that has been frozen at the moment of outer()'s invocation; the frozen function includes its full environment, along with the states of all local variables, which also means that the value of the local_var variable is successfully retained, even though the outer() function has already ceased to exist;
   - in effect, the code is fully valid, and outputs 1 to the screen;
   - the function returned during the outer() invocation is a closure.

3. Example: the following code presents a closure designed to embed its string argument within a pair of <tag> and </tag> tags (the name of the tag is passed to the closure during initialization).

```
>>> def define_mark(mark):
>>>     opening_mark = mark
>>>     closing_mark = mark.replace("<", "</")
>>>     def embed(text):
```

```
>>>         return opening_mark + text + closing_mark

>>>     return embed

>>> bold = define_mark("<b>")

>>> italic = define_mark("<i>")

>>> print(bold(italic("The Heading”)))
```

The code outputs <b><i>The Heading</i></b> to the screen.

File streams

1. A file stream is a communication channel interconnected between a program and its environment (e.g. a disk file or a terminal).

2. A standard stream is a file stream set up when the program begins execution. There are three standard streams, accessible via the sys module:

   - sys.stdin (standard input) connected to a terminal input device (e.g. a keyboard) by default (this is the channel where the input() function reads its data from)

   - sys.stdout (standard output) connected to a terminal output device (e.g. a display) by default (this is the channel where the print() function writes its data to)

   - sys.stderr (standard error output) connected to a terminal output device by default (this is the channel where Python sends its error messages to)

3. 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.

4. 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;

   - append mode: a stream opened in this mode allows writes at the end of the file.

5. A file handle is an object which encapsulates all the stream's details and provides a unified interface to all necessary stream operations and attributes. The handle object may be derived from one of the following classes:

    - RawIOBase
    - BufferedIOBase
    - TextIOBase

   All these classes are derived from the IOBase superclass.

6. The only way to obtain a functional object of the above classes is to invoke a function named open(). The object is discarded and the stream is closed when the .close() method is invoked.

   - Due to the type of the stream's contents, all the streams are divided into text and binary streams.
   - The text streams are structured in lines and contain typographical characters (letters, digits, punctuation, etc.)

7. The binary streams don't contain text, but rather a sequence of bytes of any value.

8. When the stream is open and it's told that the data in the associated file will be processed as text (or there is no such advisory at all), it switches to text mode.

9. When the stream is open and it's told to do so, it switches to binary mode, in other words, its contents are taken as-is, without any conversions.

10. The opening of the stream is performed in the following way:

    ```
    >>> stream = open(file, mode = 'r', encoding = None)
    ```

    - the open() 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 file parameter specifies the name of the file to be associated with the stream;
    - the mode parameter 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;
    - the encoding parameter 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.

11. Possible open modes and their codes passed as the second open() function argument are:

    read (coded as 'r')

    - the stream will be opened in read mode;
    - the file associated with the stream must exist and has to be readable.

    write (coded as 'w')

    - 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).

    append (coded as 'a')

    - 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 contents of the file remain untouched).

    read and update (coded as 'r+')

    - the stream will be opened in read and update mode;
    - the file associated with the stream must exist and has to be writeable.

    write and update (coded as 'w+')

    - 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; if it exists the virtual recording head will be set at the end of the file (the previous content of the file remains untouched).

12. Selecting text and binary modes is done in the following way:

    - if there is a letter b at the end of the mode string, it means that the stream is to be opened in binary mode;
    - if the mode string ends with a letter t, the stream is opened in text mode;
    - text mode is the default behaviour assumed when no binary/text mode specifier is used.

13. Closing is the operation performed on a stream and is done by an argumentless method named .close().

    Each IOError object is equipped with a property named errno (the name comes from the phrase error number) which contains an integer value representing a code of the last occurrence of the error. Some of the useful errno values come from the errno module. These are:

    - errno.EBADF → Bad file number.
    - errno.EEXIST → File exists.
    - errno.EFBIG → File too large.
    - errno.EISDIR → Is a directory.
    - errno.EMFILE → Too many open files.
    - errno.ENOENT → No such file or directory.
    - errno.ENOSPC → No space left on device.

    A complete list of errno codes is available here.

14. An error code contained in errno can be translated into text form by the os.strerror(error_code) function.

15. Example: the snippet below is a simple example of how stream errors can be handled:

    ```
    >>> import os
    >>> try:
    >>>     stream = open("c:/users/user/Desktop/file.txt", "rt")
    >>>     # Actual processing goes here.
    >>>     stream.close()
    >>> except IOError as exception:
    >>>     print("The file could not be opened:", os.strerror(exception.errno))
    ```

16. Reading from a text stream can be done character by character using the .read(chunk_size) method with an argument set to 1; the method returns a string with a length of one if the invocation is successful, or a zero-length string when there is no data to read.

Example: the following snippet shows a way in which a text file can be copied to the screen.

```
>>> import os

>>> try:

>>>     stream = open("textfile.txt", "rt")

>>>     character = stream.read(1)

>>>     while character != "":

>>>         print(character, end="")

>>>         character = s.read(1)

>>>     stream.close()

>>> except IOError as exception:

>>>     print("I/O error occurred: ", os.strerror(exception.errno))
```

Note: end-line characters are read in the same way as all other characters from the file – this is why we use the end="" clause.

17. The same task can be done line by line using the .readline() method, which returns a string with a complete line read from the file, or an empty string when there are no more lines to read in the file.

Example: the following snippet does the same job as the previous one.

```
>>> import os

>>> try:

>>>     stream = open("textfile.txt", "rt")

>>>     line = stream.readline()

>>>     while line != "":

>>>         print(line, end="")

>>>         line = stream.readline()

>>>     stream.close()
>>> except IOError as exception:

>>>     print("I/O error occurred:", os.strerror(exception.errno))
```

18. Another option is to read the file as a whole using the .readlines() method, which returns a list of strings containing all the lines read from the file.

    Example

    ```
    >>> import os

    >>> try:

    >>>     stream = open("test.py", "rt")

    >>>     lines = stream.readlines()

    >>>     for line in lines:

    >>>         print(line, end="")

    >>>     stream.close()

    >>> except IOError as exception:

    >>>     print("I/O error occurred:", strerror(exception.errno))
    ```

19. It is also possible to treat the open() function result as an iterator, which automatically generates a sequence of lines (strings) read from the file. You can expect that the .close() method is implicitly invoked when any of the file reads reaches the end of the file.

    Example

    ```
    >>> import os

    >>> try:

    >>>     for line in open("textfile.txt", "rt"):

    >>>         print(line, end="")

    >>> except IOError as exception:

    >>>     print("I/O error occurred: ", os.strerror(exception.errno))
    ```

20. Writing to the text stream is performed with the help of the .write(string) method, which sends its string argument to the associated file. No end-line character is implicitly added.

Example: the snippet below creates a text file named "newfile.txt" from scratch and fills it with 100 lines filled with 10 asterisks each.

```
>>> import os

>>> try:

>>>     stream = open("newfile.txt", "wt")

>>>     for i in range(100):

>>>         stream.write("*" * 10 + "\n")

>>>     stream.close()

>>> except IOError as exception:

>>>     print("I/O error occurred: ", os.strerror(exception.errno))
```

Note: removing the + "\n" from the function's argument causes the output file to contain only one line.

21. In order to operate with raw bytes Python has a dedicated sequence class named bytearray. Objects of the class can be used to store portions of data read in or written to a binary stream. To obtain an object of the bytearray class, one of the class constructors should be invoked, e.g.:

    ```
    >>> data = bytearray(10)
    ```

    Such an invocation creates a bytearray object named data, able to store ten bytes, and fills the array with zeros. You can treat any of the bytearray elements as integer values, and index the array in a routine way.

22. To write a bytearray's contents to a binary stream, the .write(array) method can be used. The whole of the array's contents is sent to the stream.

23. To read a bytearray's contents from a binary stream, the .readto(array) method can be used. The method returns the number of successfully read bytes and 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.

24. Example: the following code creates a file named 'file.bin', fills it with 95 bytes of values from 32 to 127, closes the stream, and reopens it in read mode in order to read its contents into a separate bytearray.

    ```
    >>> import os
    ```

```
>>> byte = bytearray(1)

>>> bytes_95 = bytearray(95)

>>> try:

>>>     out_stream = open("file.bin", "wb")

>>>     for i in range(32,127):

>>>         byte[0] = i;

>>>         out_stream.write(byte)

>>>     out_stream.close();

>>>     in_stream = open("file.bin", "rb")

>>>     read_in = in_stream.readinto(bytes_95)

>>>     in_stream.close()

>>>     print(read_in,"byte(s) read in")

>>>     for i in range(95):

>>>         print(chr(bytes_95[i]), end='')

>>> except IOError as exception:

>>>     print("I/O error occurred:", os.strerror(exception.errno))
```

25. An alternative way of reading the contents of a binary file is offered by the method named read(). The method, invoked without arguments, 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 one significant difference – it's immutable, and can be easily converted into a bytearray object.

    Note: this technique should not be used if you're not sure that the file's contents will fit the available memory. If the read()codel> method is invoked with an argument, it specifies the maximum number of bytes to be read.

    Example: the following code does the same job as the previous one.

```
>>> import os
>>> byte = bytearray(1)
>>> try:
>>>     out_stream = open("file.bin", "wb")
>>>     for i in range(32,127):
>>>         byte[0] = i;
>>>         out_stream.write(byte)
>>>     out_stream.close();
>>>     in_stream = open("file.bin", "rb")
>>>     bytes_95 = bytearray(in_stream.read())
>>>     in_stream.close()
>>>     for i in range(95):
>>>         print(chr(bytes_95[i]), end='')
>>> except IOError as exception:
>>>     print("I/O error occurred:", os.strerror(exception.errno))
```

Quiz Chapter 5

Question 1: What is the expected output of the following code?

```
>>> def f(l):
>>>     return l(-3, 3)
>>> print(f(lambda x, y: x if x > y else y))
```

A. None

B. 0

C. -3

D. 3

Question 2: What is the expected output of the following code?

```
>>> s = lambda x: 0 if x == 0 else 1 if x > 0 else -1
>>> print (s(-273.15))
```

A. -1

B. 0

C. 1

D. None

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

```
>>> l = [x for x in range(1, 10, 3) if x % 2 == 0]
>>> print(len(l))
```

A. 2

B. 8

C. 1

D. 4

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

```
>>> foo = [x for x in range(4)]

>>> spam = [x for x in foo[1:-1]]

>>> sprint(spam[1])
```

A. 2

B. 0

C. 1

D. 3

Question 5: Which of the following lines contain valid Python code? (Select two answers)

A. lambda x = x + 1

B. lambda x: None

C. lambda(x): return x + 1

D. lambda x: x + 1

Question 6: Which of the following lines contain valid Python code? (Select two answers)

A. lambda f(x, y): return x >> y

B. lambda x, y: x + y

C. lambda x, y: ‘0123456789’[x : y]

D. lambda x, y -> x ** y

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

```
>>> def f(a, b):

>>>     return a(b)

>>> print(f(lambda x: x and True, 1 > 0))
```

A. 0

B. None

C. True

D. False

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

```
>>> v = [1, 2, 3]
>>> def g(a, b, m):
>>>     return m(a, b)
>>> print(g(1, 1, lambda x, y: v[x : y + 1]))
```

[✓ A.] [2]

B. [1]

C. [3]

D. []

Question 9: What is the expected output of the following code?

```
>>> vect [“alpha”, “bravo”, “charlie”]
>>> new_vect = map(lambda s: s[0].upper(), vect)
>>> print(list(new_vect)[1])
```

A. A

B. C

[✓ C.] B

D. ABC

Question 10: What is the expected output of the following code?

```
>>> vect [“alpha”, “bravo”, “charlie”]
 >>> new_vect = filter(lambda s: s[-1].upper() in [“A”, “O”], vect)
>>> for x in new_vect:
>>>     print(x[1], end = “ ”)
```

A. RH

B. lr

C. rh

D. LR

Question 11: What is the expected output of the following code?

```
>>> def quote(quo):
>>>     def embed(str):
>>>         return quo + str + quo
>>>     return embed
>>> dblq = quote(‘ “ ‘)
>>> print(dblq(‘Jane Doe’))
```

A. Jane Doe

B. “Jane Doe”

C. ‘Jane Doe’

D. “‘Jane Doe’”

Question 12: What is the expected output of the following code?

```
>>> def inc(inc):
>>>     def do(val):
>>>         return val + inc
>>>     return do
>>> action = inc(-1)
>>> print(action(2))
```

A. 1

B. 3

C. 0

D. 2

Question 13: Which of the following statements are TRUE? (Select two answers)

A. The second argument of the open() function is an integer value

B. The print() function writes its output to the stdout stream

C. The open() function returns False when its operation fails

D. stdin, stdout, and stderr are the names of pre-opened streams

Question 14: Which of the following statements are TRUE? (Select two answers)

A. The second argument of the open() function is a string

B. Read, write, and delete are the names of file open modes.

C. Trying to write a file opened in read-only mode removes its contents.

D. The open() function raises an exception when its operation fails

Question 15: The system that allows you to diagnose input/output errors in Python is called:

A. error_number

B. errno

C. error_string

D. errcode

Question 16: Which method is used to break the connection between the file handle and a physical file?

A. disconnect()

B. lock()

C. close()

D. shutup()

Question 17: What is the expected output of the following code if there is no file named non_existing_file in the working directory/folder, and the open() function invocation is successful?

```
>>> try:
>>>     f = open(“non_existing_file”, “w”)
>>>     print(1, end = “ “)
>>>     s = f.readline()
>>>     print(2, end = “ “)
>>> except IOError as error:
>>>     print(3, end = “ “)
>>> else:
>>>     f.close()
>>>     print(4, end “ “)
```

A. 1 2 3 4

B. 1 2 4

C. 2 4

D. 1 3

Question 18: What is the expected output of the following code if the file named existing_text_file is a non-zero length text file located in the working directory and the open() function invocation is successful?

```
>>> try:
>>>     f = open(“existing_text_file”, “rt”)
>>>     spam = f.readlines()
>>>     print(len(spam))
>>>     f.close()
>>> except IOError:
>>>     print(-1)
```

A. -1

B. The length of the last line from the file

C. The number of lines contained inside the file

D. The length of the first line from the line

Answers Chapter 3

Question 1: The answers are A and B

#A

American Standard Code for Information Interchange is a character encoding standard for electronic communication established in 1963 (quite old, huh?)

#B

Unicode is a modern information encoding standard maintain by the Unicode Consortium

#C

We know nothing about it - do you?

#D

Unilang is a non-profit organization, made up of members who share a common interest in languages and linguistics.

Question 2: The answers are A and C

#A

True: e.g. the \n digraph is used to put a newline character into a string.

#B

Incorrect - As ASCII is a subset of Unicode, they have at least 127 common code points.

#C

Correct - for example, code point 32 corresponds to space (‘ ‘) in ASCII and Unicode coding standards.

#D

Incorrect - Python 3 users UTF-8 encoded source files by default.

Question 3: The answers are B and D

#A

Incorrect - As i is located before k in the Latin alphabet, its code point cannot be greater than l’s code point.

#B

Correct - In ASCII, the distance between the codepoints of upper and lower-case letters is always equal to 32, which is the code point of a space.

#C

Incorrect - In ASCII, lower-case letter code points are greater that their upper-case counterparts.

#D

Correct - ASCII alphabetic codepoints are ordered in the same order as the Latin alphabet, so ord('m') is greater by 2 than ord('k').

Question 4: The answers are A and C

#A

Correct - It's an example of a Python multi-line string, delimited by triple double quotes.

#B

Incorrect - the escape character cannot be used alone; if you want to put it in a string, you need to double it, i.e "\\"

#C

Correct - an apostrophe can be used in a string delimited by double quotes (and vice versa - double quotes can be used in a string delimited by single quotes).

#D

Incorrect - you cannot use just an apostrophe in a string delimited by single quotes. You need to precede it with \.

Question 5: The answers are C and D

#A

Error - you cannot modify immutable data (TypeError will be raised)

#B

Error - you cannot access a non-existent string character (IndexError will be raised)

#C

Correct - the operation return ‘rye’

#D

Correct - the operation return ‘y’

Question 6: The answers are A and D

#A

Correct - ‘ABCDEF’[:3] is equivalent to ‘ABCDEF’[0:3] which evaluates to ‘ABC’

#B

Incorrect - ‘FEDCBA’[-3] evaluates to ‘CBA’

#C

Incorrect - ‘AXBYCZ’[::-2] selects every second character starting from the end of the string, and evaluates to ‘ZYX’

Correct - ‘ACBYCZ’[::2] selects every second character of the string, and in effect evaluates to ‘ABC’

Question 7: The answer is D

The code is error-free and will run successful. As plane * 2 evaluates to “CessnaCessna”, the phrase c in [“e”, “a”] will return TRUE four times, hence the counter value will be the same.

Question 8: The answer is D

The source of the problem is located in the phrase: plane + 2. You can add strings to strings, and integers to integers, but adding integers to strings is prohibited - expect the TypeError exception.

Question 9: The answer is C

#A

Incorrect - the presented comparison is safe, but evaluates to TRUE as ‘AlAl’ is less than ‘Alan’.

#B

Incorrect - you cannot compare an integer to a string using an operator other than == or !=. TypeError is knocking on the door - can you hear it?

#C

Although comparing an integer value to a string always returns FALSE, the comparison itself is always safe and causes no problems.

#D

Incorrect - ‘999’ is less than ‘999999999’, so the expression evaluates to TRUE.

Question 10: The answers are B and D

#A

str(None) is a regular and valid string; it has nothing to do with the actual None.

#B

In ASCII, upper-case letters are less than their lower-case counterparts, hence the comparison evaluates to TRUE.

#C

str(None) evaluates to ‘None’, so the comparison itself is FALSE.

#D

Because an empty string which appears on the left side of the comparison (as a result of the ‘’ * 0 operation) is the least string of all, the comparison evaluates to TRUE.

Question 11: The answers are B and D

#A

The expression evaluates to FALSE because ‘’ in ‘’ is always TRUE.

#B

The expression is safe and returns TRUE

#C

As ‘xyz’ is an evident part of the ‘uvwxyz’ string, the not in operator will return FALSE.

#D

This may sound strange, but an empty string (“”) is treated as if it was an (invisible) part of every string, even an empty string. That is why the expression evaluates to TRUE.

Question 12: The answers is C and D

#A

As ‘a’ is a part of ‘abc’, and not not True is TRUE, the expression is TRUE, too.

#B

As a newline character is a part of every multi-line string, the expression evaluates to TRUE.

#C

The expression evaluates to FALSE

#D

It is worth mentioning that not x in y is not the same as x not in y - don’t forget this! Not x in y is actually evaluated as (not x) in y, which may lead to surprising results - just like the one here. As not ‘a’ evaluates to FALSE, the expression returns FALSE, too.

Question 13: The answer is A

Let’s analyze the code:

1. “Mary had 21 little sheep”.split() returns [‘Mary’, ‘had’, ’21’, ‘little’, ‘sheep’]

2. [‘Mary’, ‘had’, ’21’, ‘little’, ‘sheep’][2] returns ’21'

3. ’21’.isdigit() returns TRUE

This is the answer we’ve been looking for

Question 14: The answer is A

Let’s analyze the code:

1. ‘\”.join((“Mary”, “had”, “21”, “sheep”)) returns “Mary’had’21’sheep”

2. “Mary’had’21’sheep”[0:1] returns “M”

3. “M”.islower() returns FALSE

Question 15: The answers are A and C

#A

Correct - it is a good idea, but don’t forget that .index() may raise ValueError when the searched substring is not found. “Flash Gordon”.index(ash”) will return 2 anyway.

#B

Slicing can retrieve any part of a string, but it is not able to look for anything.

#C

Correct - this will work - let’s analyze the following example: “Flash Gordon”.rfind(“ash”) will evaluate to 2 - the place within string where “ash” starts.

#D

There is no such function as find() - you have to use the .find() method instead.

Question 16: The answers are A and C

#A

Correct - as tmp is a list of characters taken from letters, the .sort() method sorts the list in situ, while ,join() combines the list’s elements into a string.

#B

Incorrect - the sorted() function returns a list, not s string.

#C

Correct - the sorted() function returns a list of letters sorted in alphabetical order, while .join() flies the characters into one string.

#D

Incorrect - the .sort() method cannot be applied to strings, because strings are immutable.

Answers Chapter 4

Question 1: The answers are A and C

#A

Correct - a class is like a cake mold, which lets the user bake as many cakes (objects) as needed to solve a problem.

#B

Incorrect - It is inheritance.

#C

Correct - some classes are able to perform certain usable actions without the need to create class objects, but you cannot create an object of a non-existent class.

#D

Incorrect - you cannot have more than one class of a given name. Polymorphism allows you to differentiate between the behavior of objects, not the names of classes.

Question 2: The answers are A and C

#A

Correct - In Python, the constructor is a function called __init__(). You can use it to initialize the new object's state.

#B

Incorrect - a superclass is located above its class.

#C

Correct - encapsulation lets you hide sensitive object components and protect them from unauthorized modification.

#D

Incorrect - In Python, each, class object can have a different set of attributes.

Question 3: The answer is D

The code is error-free and it will run successfully. As label is a class variable, each class object sees the same variable value, hence the code prints TRUE.

Question 4: The answer is B

The code is error-free and it will run successfully. The spam variable is a class variable, while the class constructor appends its string argument followed by ';' to the variable. In effect, spam will contain the string Capacitors; Transistors;

Question 5: The answers are B and C

#A

It is equal to 0, actually, as the object's directory is empty.

#B

An object's __dict__ is not the same as __dict__ of its home class. Moreover, the binder object's __dict__ is empty as the dispose() function has been executed. Hence, these two lengths cannot be equal.

#C

stamps is a class variable, and it is included in the class's __dict__

#D

Note that dispose() has removed the stuff from the object's dict. It is not in the directory anymore.

Question 6: The answers are B and C

#A

The two dictionaries store completely different sets of data. Their lengths may be equal, but it's unlikely, as the object's __dict__ contains just one element: the instance_var variable.

#B

As class methods are included in the class's __dict__, and the method of the specified name exists, the left side of the comparison is definitely not None.

#C

It may look strange, but __dict__ is included in __dict__, just like any other component of the class. Nothing unusual.

#D

Methods are not included in an object’s __dict__. KeyError will knock on your door. Don’t be surprised.

Question 7: The answers are A and C

#A

It is an example of Python’s name mangling using the product._BluePrint__element syntax we are able to bypass the variable’s privacy using its object name, and to access its value.

#B

It won’t work - a variable of this name doesn’t exist, and name mangling is not properly used.

#C

We’re dealing here with an example of name mangling using the BluePrint._BluePrint_element syntax we are able to bypass the variable’s privacy using its home class name, and to access its value.

#D

If you want to raise the NameError exception, this is a pretty good way to do it.

Question 8: The answer is B

#A

It would work if you wanted to access a class variable, but private functions cannot be accessed in this way.

#B

This is how name mangling helps you to bypass the restrictions established by Python’s privacy policy. Using the product._BluePrint__action() syntax makes the invocation possible.

#C

This is raise AttributeError

#D

You cannot invoke a private method in this way.

Question 9: The answer is A

#A

Perfect! This is how it should be done!

#B

Error - the self parameter doesn’t exist, nor is the rack variable accessible.

#C

Error - the method lacks the self parameter.

#D

Error - the method tries to use a non-existent variable name.

Question 10: The answers are B and C

#A

The TypeError exception will be raised

#B

This just works. Nothing more, nothing less.

#C

Correct, as the function is invoked at the class level, the target object has to be passed explicitly as an argument.

#D

The NameError exception will be raised.

Question 11: The answer is B

The three most important aspects are listed here:

- it is a tuple;
- it contains superclass data;
- only direct superclasses are included.

Missing any of the above would make the sentence incorrect.

Question 12: The answer is B

- True - the holder object has the Token attribute, because the attribute is derived from the superclass (it is a class variable).
- False - the Ceil class does not have the set_token() function as it shows up in the Floor subclass only.

Question 13: The answer is A

Let's dive into each of the expressions:

  - busn_a is an instance of the Economy class as it is an object of Economy's subclass -> True
  - econ_a is not an instance of the Business class as it is an object of Business's superclass -> False
- True and False -> False;
  - busn_a and busn_b actually refer to the same object, hence busn_b is busn_a evaluates to True;
  - econ_a and econ_b refer to different objects, thus, econ_a is busn_b evaluates to False;
- True or False -> True

Question 14: The answers are A and D

#A

True: this attribute's value has been defined at the Middle class level, short's superclass.

#B

False: an object cannot be an instance of its home class's subclass.

#C

False: It is 3 actually (it is set at the Middle class level)

#D

True: This attribute value has been defined at the Middle class level, average's home class.

Question 15: The answers are A and C

#A

As d is an object of Beta's subclass, it is an instance of Beta and the isinstance() invocation returns TRUE.

#B

Alpha is not Delta's direct superclass, hence the expression evaluates to FALSE

#C

According to Python's Method Resolution Order (MRO), say() in the d object comes from the Beta class definition, hence the comparison results in TRUE.

#D

The value presented above comes from the Alpha class definition, but it has been overridden by Beta, and that is why the expression evaluates to FALSE.

Question 16: The answers are B and D

#A

b and d are different objects of different classes - the expression results in FALSE

#B

Gamma is one of the direct superclass of Delta, so it is included in __bases__ - that is why this expression evaluates to TRUE.

#C

According to Python's Method Resolution Order (MRO), the d() function visible in Delta class object comes from Beta class definition - that is why the expression is FALSE

#D

As d is an object of a class which is a subclass (indirect) of Alpha, it is an instance of Alpha, and that is why isinstance() returns TRUE.

Question 17: The answers are B and D

#A

A constructor cannot return a result

#B

Constructors are allowed to raise exceptions; this constructor is correct

#C

The self parameter (or its equivalent) is missing; the constructor is incorrect

#D

All requirement are met; this constructor is correct

Question 18: The answers are A and D

#A

Although the implicit constructor invocation doesn't specify any arguments, the default parameter value is used and True is printed.

#B

The constructor lacks the self parameter - the code won't run

#C

As the subclass defines its own constructor, the superclass constructor won't be invoked, and False is the only output visible on the screen.

#D

As B doesn't define its own constructor, the superclass's constructor is invoked and, in effect, True is printed to the screen.

Question 19: The answer is A

Polymorphism and the fact that the A class has overridden the show_ID() function's definition, means that the output will read Alpha, not anonymous.

Question 20: The answer is C

Note that:

1. Fixed_Wing doesn't override Aircraft's start() method, so the invocation of start() will output the following string: 'default';

2. The Rotor_Craft defines its own start() function, which polymorphically overrides the function of the same name defined in the superclass, hence Rotor_Craft's start() function prints the following string" 'spin'

In effect, the string 'default spin' appears on the screen.

Question 21: The answer is C

This is the only acceptable answer - nothing else will work.

Question 22: The answer is A

The special variable __name__ contains the string '__main__' when the code is run a standalone program, or a module name if a Python file is used as a module. No other listed variable exists.

Question 23: The answers are A and D

#A

The declaration follows the Method Resolution Order rules, and Python will accept this.

#B

The declaration breaks the Method Resolution Order rules - the TypeError exception will be raised.

#C

The declaration breaks the Method Resolution Order rules - the TypeError exception will be raised.

#D

The Method Resolution Order rules are met - the declaration is correct.

Question 24: The answer is D

According to the Method Resolution Order (MRO), the Bottom class will be searched through first in order to get the string representation of the object, so the Bottom class's __str__() function will be invoked - that is why 3 is printed.

Answers Chapter 4

Question 1: The answer is D

The lambda function used by the code finds the greater of its two arguments (can you can see it?). the function is passed as an argument to f(), which invokes the lambda with two argument: -3 and 3. In effect, 3 appears on the screen.

Question 2: The answer is A

Note that the lambda function defined in the code implements the signum function (mathematicians know it as sgn(x), which returns:

- -1 if its argument is less than zero;
- 0 if the argument is zero;
- 0r 1 when the argument is greater than zero.

This is why the code print -1 to the screen.

Question 3: The answer is C

The list comprehension embedded in the code produces just one value: 4. That is why the list's length is 1

Question 4: The answer is A

Let's analyze the code's execution:

- The first comprehension produces a list that contains 0, 1, 2, and 3;
- The second uses its two central values only: 1 and 2;
- That is why spam[1] is equal to 2.

Question 5: The answers are B and D

#A

The declaration lacks the argument list. The SyntaxError exception will visit you soon.

#B

This lambda returns None regardless of its argument's value - not very useful, but correct.

#C

Parentheses are not welcome here and SyntaxError is inevitable.

#D

This lambda returns its argument incremented by one.

Question 6: The answers are B and C

#A

The code is closer to a regular function declaration than that of a lambda. Anyway, the SyntaxError exception will be printed.

#B

This lambda returns the sum of its two arguments.

#C

This lambda slices the string that contains a complete set of decimal digits based on its lambda's argument values.

#D

The -> digraph is misused here - Python won't like it and will respond with SyntaxError.

Question 7: The answer is C

Let's analyze the case:

- The f() function treats its first argument as a function, and the second as its argument;
- The lambda finds the Boolean conjunction of True and its argument; In short, it returns True when the argument can be identified as True, e.g, when it is a non-zero number;
- Invoking the lambda with 1 as its argument will result in True;
- This is what we'll see on the screen.

Question 8: The answer is A

Let's analyze the case:

- The lambda takes its two arguments and uses them as limits for slicing the v list;
- The g() function uses its third argument as a function to invoke while the first two arguments are passed to it;

- The slice v[1:1] evaluates to [2]…
- … and this is what we'll see on the screen.

Question 9: The answer is C

Let's analyze the execution step by step:

- The lambda function returns the first argument's character in upper case.
- The lambda function is applied to all vect's elements (this is how the map() function works);
- The list built of new_vect will contain the 'B' character inside the element indexed by 1;
- This is the only output sent to the screen.

Question 10: The answer is B

The filter() function will drop all vect's elements whose last character in upper case is neither "A" nor "O". As both "alpha" and "bravo" meet the requirement, they will be included in new_vect. Later, the for loop prints the second character of the vector it iterates through, so we'll see lr on the screen.

Question 11: The answer is B

The code is an example of how the closure concept is implemented in Python. The quote() function produces a closure which surrounds its string argument with a character passed earlier to quote(). As dblq() is dedicated to putting its argument inside double quotes, the expected output is "Jane Doe".

Question 12: The answer is A

The action() function is a closure built with the help of inc(), which is able to generate functions that can increment its argument with a previously defined value. As there is the inc(-1) invocation in the snippet, action() will reduce its argument by 1, returning 1 as the result.

Question 13: The answers are B and D

#A

No, it isn't. It is a string specifying the open mode.

#B

This is exactly what we expect from print()

#C

The open() function returns a file object, not a Boolean, and raises an exception when failure occurs.

#D

Let’s add that these names were introduced by the very first version of Unix and are universally honored in all modern operating systems.

Question 14: The answers are A and D

#A

Yes, it is. We use it to describe the open mode.

#B

While read and write look good., delete doesn’t fit the company. Append is the missing link.

#C

No, it doesn’t. It is illegal and will raise the io.UnsupportedOperation exception.

#D

Yes, it does. This is how we learn that a failure has occurred.

Question 15: The answer is B

This name was introduced by the “C” language standard library implemented a long time ago for Unix, and is widely used now by many programming frameworks. All other names are made up - sorry!

Question 16: The answer is C

A file which has been opened should be closed

Question 17: The answer is D

Trying to read from a file opened in write mode will end in disaster. An exception is raised, and the print(2, end = “ “) invocation cannot be reached. In effect, the else branch is bypassed, too. Thi is why the only digits appearing on screen are 1 and 3.

Question 18: The answer is C

The .readlines() method tries to read the entire contents of the file and return it as a list of lines. This means that the length of the returned list is equal to the number of lines contained in the file.