1.	What does the following code do? def a(b, c, d): pass	Answer: defines a function, which does nothing
2.	What gets printed? Assuming python version 3.x print(type(1/2))	Answer: <class 'float'> / <type 'int'> in 2.7
3.	What is the output of the following code? print(type([1,2]))	Answer: <class 'list'> / <type 'list'> in 2.7
4.	What gets printed? def f(): pass print(type(f()))	Answer: <class 'NoneType'> / <type 'NoneType'> in 2.7
5.	What should the below code print? print(type(1J))	Answer: <class 'complex'> / <type 'complex'> in 2.7
6.	What is the output of the following code? print(type(lambda:None))	Answer: <class 'function'> / <type 'function'> in 2.7
7.	What is the output of the below program? a = [1,2,3,None,(),[],] print(len(a))	Answer: 6
8.	What gets printed? d = lambda p: p * 2 t = lambda p: p * 3 x = 2 x = d(x) x = t(x) x = d(x) print(x)	Answer: 24

9. What gets printed?

```
x = 4.5
y = 2
print(x//y)
print(x/y)
```

Answer: 2.0
2.25

---

10. What gets printed?

```
nums = set([1,1,2,3,3,3,4])
print(len(nums))
```

Answer: 4

---

11. What gets printed?

```
x = True
y = False
z = False
if x or y and z:
print("yes")
else:
print("no")
```

Answer: "yes"

---

12. What gets printed?

```
x = True
y = False
z = False
if not x or y:
print 1
elif not x or not y and z:
print 2
elif not x or y or not y and x:
print 3
else:
print 4
```

Answer: 3

---

13. If PYTHONPATH is set in the environment, which directories are searched for modules?
A) PYTHONPATH directory
B) current directory
C) home directory
D) installation dependent default path

Answer: A, B, D
Explanation: First is the current directory, then is the PYTHONPATH directory if set, then is the installation dependent default path

---

14. What gets printed?

print(r"\nwoow")

Answer: \nwoow

---

15. What gets printed?

print('elo' 'elo')

Answer: eloelo

---

16. What gets printed?
print("\x48\x49!")

Answer: HI!

---

17. What gets printed?

print(0xA + 0xa)

Answer: 20

---

18. What gets printed?
class parent:
def __init__(self, param):
self.v1 = param
class child(parent):
def __init__(self, param):
self.v2 = param
obj = child(11)
print("{} {}".format(obj.v1, obj.v2))

Answer: AttributeError: child instance has no attribute 'v1'. self.v1 was never created as a variable since the parent __init__ was not explicitly called

---

19. What gets printed?
k1 = {'e', 'd'}
k2 = {'e':1}
print(type(k1), type(k2))

Answer: <class 'set'>, <class 'dict'>

---

20. What gets printed?

class Account:
def __init__(self, id):
self.id = id
id = 666
acc = Account(123)
print(acc.id)

Answer: 123

---

21. What gets printed?
name = 'snow storm'
print(name[6:8])

Answer: 'to'

22. Which is the correct operator for power(x^y)?
a) X^y
b) X**y
c) X^^y
d) None of the mentioned

Answer: b
Explanation: In python, power operator is x**y i.e. 2**3=8.

23. Which one of these is floor division?
a) /
b) //
c) %
d) None of the mentioned

Answer: b
Explanation: When both of the operands are integer then python chops out the fraction part and gives you the round off value, to get the accurate answer use floor division. This is floor division. For ex, 5/2 = 2.5 but both of the operands are integer so answer of this expression in python is 2.To get the 2.5 answer, use floor division.

24. What is the order of precedence in python?
i) Parentheses
ii) Exponential
iii) Division
iv) Multiplication
v) Addition
vi) Subtraction
a) i,ii,iii,iv,v,vi
b) ii,i,iii,iv,v,vi
c) ii,i,iv,iii,v,vi
d) i,ii,iii,iv,vi,v

Answer: a
Explanation: For order of precedence, just remember this PEDMAS.

25. What is answer of this expression, 22 % 3 is?
a) 7
b) 1
c) 0
d) 5

Answer: b
Explanation: Modulus operator gives remainder. So, 22%3 gives 1 remainder.

26. You can perform mathematical operation on String?
a) True
b) False

Answer: b
Explanation: You can’t perform mathematical operation on string even if string looks like integers.

27. Operators with the same precedence are evaluated in which manner?
a) Left to Right
b) Right to Left

Answer: a
Explanation: None.

---

28. What is the output of this expression, 3*1**3?
a) 27
b) 9
c) 3
d) 1

Answer: c
Explanation: First this expression will solve 1**3 because exponential have higher precedence than multiplication, so 1**3 = 1 and 3*1 = 3. Final answer is 3.

---

29. Which one of the following have the same precedence?
a) Addition and Subtraction
b) Multiplication and Division
c) a and b
d) None of the mentioned

Answer: c
Explanation: None.

---

30. Int(x) means variable x is converted to integer.
a) True
b) False

Answer: a
Explanation: None.

---

31. Which one of the following have the highest precedence in the expression?
a) Exponential
b) Addition
c) Multiplication
d) Parentheses

Answer: d
Explanation: None.

---

32. Is Python case sensitive when dealing with identifiers?
a) yes
b) no
c) sometimes only
d) none of the mentioned

Answer: a
Explanation: Case is always significant.

---

33. What is the maximum possible length of an identifier?
a) 31 characters
b) 63 characters
c) 79 characters
d) none of the mentioned

Answer: d
Explanation: Identifiers can be of any length.

---

34. Which of the following is invalid?
a) _a = 1
b) __a = 1
c) __str__ = 1
d) none of the mentioned

Answer: d
Explanation: All the statements will execute successfully but at the cost of reduced readability.

---

35. Which of the following is an invalid variable?
a) my_string_1
b) 1st_string
c) foo
d) _

Answer: b
Explanation: Variable names should not start with a number.

---

36. Why are local variable names beginning with an underscore discouraged?
a) they are used to indicate a private variables of a class
b) they confuse the interpreter
c) they are used to indicate global variables
d) they slow down execution

Answer: a
Explanation: As Python has no concept of private variables, leading underscores are used to indicate variables that must not be accessed from outside the class.

---

37. Which of the following is not a keyword?
a) eval
b) assert
c) nonlocal
d) pass

Answer: a
Explanation: eval can be used as a variable.

---

38. All keywords in Python are in
a) lower case
b) UPPER CASE
c) Capitalized
d) none

Answer: d
Explanation: True, False and None are capitalized while the others are in lower case.

---

39. Which of the following is true for variable names in Python?
a) unlimited length
b) all private members must have leading and trailing underscores
c) underscore and ampersand are the only two special charaters allowed
d) none

Answer: a
Explanation: Variable names can be of any length.

---

40. Which of the following is an invalid statement?
a) abc = 1,000,000
b) a b c = 1000 2000 3000
c) a,b,c = 1000, 2000, 3000
d) a_b_c = 1,000,000

Answer: b
Explanation: Spaces are not allowed in variable names.

---

41. Which of the following cannot be a variable?
a) __init__
b) in
c) it
d) on

Answer: b
Explanation: in is a keyword.

---

42. Which module in Python supports regular expressions?
a) re
b) regex
c) pyregex
d) none of the mentioned

Answer: a
Explanation: re is a part of the standard library and can be imported using: import re.

---

43. Which of the following creates a pattern object?
a) re.create(str)
b) re.regex(str)
c) re.compile(str)
d) re.assemble(str)

Answer: c
Explanation: It converts a given string into a pattern object.

---

44. What does the function re.match do?
a) matches a pattern at the start of the string
b) matches a pattern at any position in the string
c) such a function does not exist
d) none of the mentioned

Answer: a
Explanation: It will look for the pattern at the beginning and return None if it isn’t found.

---

45. What does the function re.search do?
a) matches a pattern at the start of the string
b) matches a pattern at any position in the string
c) such a function does not exist
d) none of the mentioned

Answer: b
Explanation: It will look for the pattern at any position in the string.

---

46. What is the output of the following?
sentence = 'we are humans'
matched = re.match(r'(.*) (.*?) (.*)', sentence)
print(matched.groups())
a) (‘we’, ‘are’, ‘humans’)
b) (we, are, humans)
c) (‘we’, ‘humans’)
d) ‘we are humans’

Answer: a
Explanation: This function returns all the subgroups that have been matched.

---

47. What is the output of the following?
sentence = 'we are humans'
matched = re.match(r'(.*) (.*?) (.*)', sentence)
print(matched.group())
a) (‘we’, ‘are’, ‘humans’)
b) (we, are, humans)
c) (‘we’, ‘humans’)
d) ‘we are humans’

Answer: d
Explanation: This function returns the entire match.

48. What is the output of the following?
sentence = 'we are humans'
matched = re.match(r'(.*) (.*?) (.*)', sentence)
print(matched.group(2))
a) ‘are’
b) ‘we’
c) ‘humans’
d) ‘we are humans’

Answer: c
Explanation: This function returns the particular subgroup.

49. What is the output of the following?
sentence = 'horses are fast'
regex = re.compile('(?P\w+) (?P\w+) (?P\w+)')
matched = re.search(regex, sentence)
print(matched.groupdict())
a) {‘animal’: ‘horses’, ‘verb’: ‘are’, ‘adjective’: ‘fast’}
b) (‘horses’, ‘are’, ‘fast’)
c) ‘horses are fast’
d) ‘are’

Answer: a
Explanation: This function returns a dictionary that contains all the mathches.

50. What is the output of the following?
sentence = 'horses are fast'
regex = re.compile('(?P\w+) (?P\w+) (?P\w+)')
matched = re.search(regex, sentence)
print(matched.groups())
a) {‘animal’: ‘horses’, ‘verb’: ‘are’, ‘adjective’: ‘fast’}
b) (‘horses’, ‘are’, ‘fast’)
c) ‘horses are fast’
d) ‘are’

Answer: b
Explanation: This function returns all the subgroups that have been matched.

51. What is the output of the following?
sentence = 'horses are fast'
regex = re.compile('(?P\w+) (?P\w+) (?P\w+)')
matched = re.search(regex, sentence)
print(matched.group(2))
a) {‘animal’: ‘horses’, ‘verb’: ‘are’, ‘adjective’: ‘fast’}
b) (‘horses’, ‘are’, ‘fast’)
c) ‘horses are fast’
d) ‘are’

Answer: d
Explanation: This function returns the particular subgroup.

---

52. What is the output of print 0.1 + 0.2 == 0.3?
a) True
b) False
c) Machine dependent
d) Error

Answer: b
Explanation: Neither of 0.1, 0.2 and 0.3 can be represented accurately in binary. The round off errors from 0.1 and 0.2 accumulate and hence there is a difference of 5.5511e-17 between (0.1 + 0.2) and 0.3.

---

53. Which of the following is not a complex number?
a) k = 2 + 3j
b) k = complex(2, 3)
c) k = 2 + 3l
d) k = 2 + 3J

Answer: c
Explanation: l (or L) stands for long.

---

54. What is the type of inf?
a) Boolean
b) Integer
c) Float
d) Complex

Answer: c
Explanation: Infinity is a special case of floating point numbers. It can be obtained by float(‘inf’).

---

55. What does ~4 evaluate to?
a) -5
b) -4
c) -3
d) +3

Answer: a
Explanation: ~x is equivalent to -(x+1).

---

56. What does ~~~~~~5 evaluate to?
a) +5
b) -11
c) +11
d) -5

Answer: a
Explanation: ~x is equivalent to -(x+1).

---

57. Which of the following is incorrect?
a) x = 0b101
b) x = 0x4f5
c) x = 19023
d) x = 03964

Answer: d
Explanation: Numbers starting with a 0 are octal numbers but 9 isn’t allowed in octal numbers.

---

58. What is the result of cmp(3, 1)?
a) 1
b) 0
c) True
d) False

Answer: a
Explanation: cmp(x, y) returns 1 if x > y, 0 if x == y and -1 if x < y.

---

59. What is the output of the following?
x = ['ab', 'cd']
for i in x:
x.append(i.upper())
print(x)
a) [‘AB’, ‘CD’]
b) [‘ab’, ‘cd’, ‘AB’, ‘CD’]
c) [‘ab’, ‘cd’]
d) none of the mentioned

Answer: d
Explanation: The loop does not terminate as new elements are being added to the list in each iteration.

---

60. What is the output of the following?
i = 1
while True:
if i%3 == 0:
break
print(i)
i + = 1
a) 1 2
b) 1 2 3
c) error
d) none of the mentioned

Answer: c
Explanation: SyntaxError, there shouldn’t be a space between + and = in +=.

---

61. What is the output of the following?

```
i = 1
while True:
if i%0O7 == 0:
break
print(i)
i += 1
```

a) 1 2 3 4 5 6
b) 1 2 3 4 5 6 7
c) error
d) none of the mentioned

Answer: a
Explanation: Control exits the loop when i becomes 7.

---

62. What is the output of the following?

```
i = 5
while True:
if i%0O11 == 0:
break
print(i)
i += 1
```

a) 5 6 7 8 9 10
b) 5 6 7 8
c) 5 6
d) error

Answer: b
Explanation: 0O11 is an octal number.

---

63. What is the output of the following?

```
i = 5
while True:
if i%0O9 == 0:
break
print(i)
i += 1
```

a) 5 6 7 8
b) 5 6 7 8 9
c) 5 6 7 8 9 10 11 12 13 14 15 ….
d) error

Answer: d
Explanation: 9 isn’t allowed in an octal number.

---

64. What is the output of the following?

```
i = 1
while True:
if i%2 == 0:
break
print(i)
i += 2
```

a) 1
b) 1 2
c) 1 2 3 4 5 6 …
d) 1 3 5 7 9 11 …

Answer: d
Explanation: The loop does not terminate since i is never an even number.

---

65. What is the output of the following?

```
i = 2
while True:
if i%3 == 0:
break
print(i)
i += 2
```

a) 2 4 6 8 10 …
b) 2 4
c) 2 3
d) error

Answer: b
Explanation: The numbers 2 and 4 are printed. The next value of i is 6 which is divisible by 3 and hence control exits the loop.

---

66. What is the output of the following?

```
i = 1
while False:
if i%2 == 0:
break
print(i)
i += 2
```

a) 1
b) 1 3 5 7 …
c) 1 2 3 4 …
d) none of the mentioned

Answer: d
Explanation: Control does not enter the loop because of False.

---

67. What is the output of the following?
True = False
while True:
print(True)
break
a) True
b) False
c) None
d) none of the mentioned

Answer: d
Explanation: SyntaxError, True is a keyword and it's value cannot be changed.

---

68. What is the output of the following?
i = 0
while i < 3:
print(i)
i += 1
else:
print(0)
a) 0 1 2 3 0
b) 0 1 2 0
c) 0 1 2
c) error

Answer: b
Explanation: The else part is executed when the condition in the while statement is false.

---

69. What is the output of the following?
x = 'abcdef'
while i in x:
print(i, end=' ')
a) a b c d e f
b) abcdef
c) i i i i i i …
d) error

Answer: d
Explanation: NameError, i is not defined.

---

70. What is the output of the following?
x = 'abcdef'
i = 'i'
while i in x:
print(i, end=' ')

a) no output
b) i i i i i i …
c) a b c d e f
d) abcdef

Answer: a
Explanation: "i" is not in "abcdef".

---

71. What is the output of the following?
x = 'abcdef'
i = 'a'
while i in x:
print(i, end = ' ')
a) no output
b) i i i i i i …
c) a a a a a a …
d) a b c d e f

Answer: c
Explanation: As the value of i or x isn’t changing, the condition will always evaluate to True.

---

72. What is the output of the following?
x = 'abcdef'
i = 'a'
while i in x:
print('i', end = ' ')
a) no output
b) i i i i i i …
c) a a a a a a …
d) a b c d e f

Answer: b
Explanation: As the value of i or x isn’t changing, the condition will always evaluate to True.

---

73. What is the output of the following?
x = 'abcdef'
i = 'a'
while i in x:
x = x[:-1]
print(i, end = ' ')
a) i i i i i i
b) a a a a a a
c) a a a a a
d) none of the mentioned

Answer: b
Explanation: The string x is being shortened by one charater in each iteration.

---

74. What is the output of the following?
x = 'abcdef'
i = 'a'
while i in x[:-1]:
print(i, end = ' ')
a) a a a a a
b) a a a a a a
c) a a a a a a …
d) a

Answer: c
Explanation: String x is not being altered and i is in x[:-1].

---

75. What is the output of the following?
x = 'abcdef'
i = 'a'
while i in x:
x = x[1:]
print(i, end = ' ')
a) a a a a a a
b) a
c) no output
d) error

Answer: b
Explanation: The string x is being shortened by one charater in each iteration.

---

76. What is the output of the following?
x = 'abcdef'
i = 'a'
while i in x[1:]:
print(i, end = ' ')
a) a a a a a a
b) a
c) no output
d) error

Answer: c
Explanation: i is not in x[1:].

---

77. What is the output of the following?
x = 'abcd'
for i in x:
print(i.upper())
a) a b c d
b) A B C D
c) a B C D
d) error

Answer: b
Explanation: The instance of the string returned by upper() is being printed.

---

78. What is the output of the following?
x = 'abcd'
for i in range(x):
print(i)
a) a b c d
b) 0 1 2 3
c) error
d) none of the mentioned

Answer: c
Explanation: range(str) is not allowed.

---

79. What is the output of the following?
x = 'abcd'
for i in range(len(x)):
print(i)

a) a b c d
b) 0 1 2 3
c) error
d) none of the mentioned

Answer: b
Explanation: i takes values 0, 1, 2 and 3.

---

80. What is the output of the following?
x = 'abcd'
for i in range(len(x)):
print(i.upper())
a) a b c d
b) 0 1 2 3
c) error
d) none of the mentioned

Answer: c
Explanation: Objects of type int have no attribute upper().

---

81. What is the output of the following?
x = 'abcd'
for i in range(len(x)):
i.upper()
print (x)
a) a b c d
b) 0 1 2 3
c) error
d) none of the mentioned

Answer: c
Explanation: Objects of type int have no attribute upper().

---

82. What is the output of the following?
x = 'abcd'
for i in range(len(x)):
x[i].upper()
print (x)
a) abcd
b) ABCD
c) error
d) none of the mentioned

Answer: a
Explanation: Changes do not happen in-place, rather a new instance of the string is returned.

---

83. What is the output of the following?
x = 'abcd'
for i in range(len(x)):
i[x].upper()
print (x)
a) abcd
b) ABCD
c) error
d) none of the mentioned

Answer: c
Explanation: Objects of type int aren’t subscriptable.

---

84. What is the output of the following?
x = 'abcd'
for i in range(len(x)):
x = 'a'
print(x)
a) a
b) abcd abcd abcd
c) a a a a
d) none of the mentioned

Answer: c
Explanation: range() is computed only at the time of entering the loop.

---

85. What is the output of the following?
x = 'abcd'
for i in range(len(x)):
print(x)
x = 'a'
a) a
b) abcd abcd abcd abcd
c) a a a a
d) none of the mentioned

Answer: d
Explanation: abcd a a a is the output as x is modified only after ‘abcd’ has been printed once.

---

86. What is the output of the following?
d = {0: 'a', 1: 'b', 2: 'c'}
for i in d:
print(i)
a) 0 1 2
b) a b c
c) 0 a 1 b 2 c
d) none of the mentioned

Answer: a
Explanation: Loops over the keys of the dictionary.

---

87. What is the output of the following?
d = {0: 'a', 1: 'b', 2: 'c'}
for x, y in d:
print(x, y)
a) 0 1 2
b) a b c
c) 0 a 1 b 2 c
d) none of the mentioned

Answer: d
Explanation: Error, objects of type int aren't iterable.

---

88. What is the output of the following?
d = {0: 'a', 1: 'b', 2: 'c'}
for x, y in d.items():
print(x, y)

a) 0 1 2
b) a b c
c) 0 a 1 b 2 c
d) none of the mentioned

Answer: c
Explanation: Loops over key, value pairs.

---

89. What is the output of the following?
d = {0: 'a', 1: 'b', 2: 'c'}
for x in d.keys():
print(d[x])
a) 0 1 2
b) a b c
c) 0 a 1 b 2 c
d) none of the mentioned

Answer: b
Explanation: Loops over the keys and prints the values.

---

90. What is the output of the following?
d = {0: 'a', 1: 'b', 2: 'c'}
for x in d.values():
print(x)
a) 0 1 2
b) a b c
c) 0 a 1 b 2 c
d) none of the mentioned

Answer: b
Explanation: Loops over the values.

---

91. What is the output of the following?
d = {0: 'a', 1: 'b', 2: 'c'}
for x in d.values():
print(d[x])
a) 0 1 2
b) a b c
c) 0 a 1 b 2 c
d) none of the mentioned

Answer: d
Explanation: Causes a KeyError.

---

92. What is the output of the following?
d = {0, 1, 2}
for x in d.values():
print(x)

a) 0 1 2
b) None None None
c) error
d) none of the mentioned

Answer: c
Explanation: Objects of type set have no attribute values.

---

93. What is the output of the following?
d = {0, 1, 2}
for x in d:
print(x)
a) 0 1 2
b) {0, 1, 2} {0, 1, 2} {0, 1, 2}
c) error
d) none of the mentioned

Answer: a
Explanation: Loops over the elements of the set and prints them.

---

94. What is the output of the following?
d = {0, 1, 2}
for x in d:
print(d.add(x))
a) 0 1 2
b) 0 1 2 0 1 2 0 1 2 …
c) None None None
d) none of the mentioned

Answer: c
Explanation: Variable x takes the values 0, 1 and 2. set.add() returns None which is printed.

---

95. What is the output of the following?
for i in range(0):
print(i)
a) 0
b) (nothing is printed)
c) error
d) none of the mentioned

Answer: b
Explanation: range(0) is empty.

---

96. What is the output of the following?
for i in range(int(2.0)):
print(i)
a) 0.0 1.0
b) 0 1
c) error
d) none of the mentioned

Answer: b
Explanation: range(int(2.0)) is the same as range(2).

---

97. What is the output of the following?
for i in range(float('inf')):
print (i)
a) 0.0 0.1 0.2 0.3 …
b) 0 1 2 3 …
c) 0.0 1.0 2.0 3.0 …
d) none of the mentioned

Answer: d
Explanation: Error, objects of type float cannot be interpreted as an integer.

---

98. What is the output of the following?
for i in range(int(float('inf'))):
print (i)

a) 0.0 0.1 0.2 0.3 …
b) 0 1 2 3 …
c) 0.0 1.0 2.0 3.0 …
d) none of the mentioned

Answer: d
Explanation: OverflowError, cannot convert float infinity to integer.

---

99. What is the output of the following?
for i in [1, 2, 3, 4][::-1]:
print (i)
a) 1 2 3 4
b) 4 3 2 1
c) error
d) none of the mentioned

Answer: b
Explanation: [::-1] reverses the list.

---

100. What is the output of the following?
for i in ''.join(reversed(list('abcd'))):
print (i)
a) a b c d
b) d c b a
c) error
d) none of the mentioned

Answer: b
Explanation: ”.join(reversed(list(‘abcd’))) reverses a string.

---

101. What is the output of the following?
for i in 'abcd'[::-1]:
print (i)
a) a b c d
b) d c b a
c) error
d) none of the mentioned

Answer: b
Explanation: [::-1] reverses the string.

---

102. What is the output of the following?
for i in '':
print (i)
a) None
b) (nothing is printed)
c) error
d) none of the mentioned

Answer: b
Explanation: The string does not have any character to loop over.

---

103. What is the output of the following?
x = 2
for i in range(x):
x += 1
print (x)
a) 0 1 2 3 4 …
b) 0 1
c) 3 4
d) 0 1 2 3

Answer: c
Explanation: Variable x is incremented and printed twice.

---

104. What is the output of the following?
x = 2
for i in range(x):
x -= 2
print (x)
a) 0 1 2 3 4 …
b) 0 -2
c) 0
d) error

Answer: b
Explanation: The loop is entered twice.

---

105. What is the output of the following?
for i in range(5):
if i == 5:
break
else:
print(i)
else:
print('Here')
a) 0 1 2 3 4 Here
b) 0 1 2 3 4 5 Here
c) 0 1 2 3 4
d) 1 2 3 4 5

Answer: a
Explanation: The else part is executed if control doesn't break out of the loop.

---

106. What is the output of the following?
x = (i for i in range(3))
for i in x:
print(i)
a) 0 1 2
b) error
c) 0 1 2 0 1 2
d) none of the mentioned

Answer: a
Explanation: The first statement creates a generator object.

---

107. What is the output of the following?
x = (i for i in range(3))
for i in x:
print(i)
for i in x:
print(i)

a) 0 1 2
b) error
c) 0 1 2 0 1 2
d) none of the mentioned

Answer: a
Explanation: We can loop over a generator object only once.

---

108. What is the output of the following?
string = 'my name is x'
for i in string:
print (i, end=', ')
a) m, y, , n, a, m, e, , i, s, , x,
b) m, y, , n, a, m, e, , i, s, , x
c) my, name, is, x,
d) error

Answer: a
Explanation: Variable i takes the value of one character at a time.

---

109. What is the output of the following?
string = 'my name is x'
for i in string.split():
print (i, end=', ')
a) m, y, , n, a, m, e, , i, s, , x,
b) m, y, , n, a, m, e, , i, s, , x
c) my, name, is, x,
d) error

Answer: c
Explanation: Variable i takes the value of one word at a time.

---

110. What is the output of the following?
a = [0, 1, 2, 3]
for a[-1] in a:
print(a[-1])
a) 0 1 2 3
b) 0 1 2 2
c) 3 3 3 3
d) error

Answer: b
Explanation: The value of a[-1] changes in each iteration.

---

111. What is the output of the following?
a = [0, 1, 2, 3]
for a[0] in a:
print(a[0])
a) 0 1 2 3
b) 0 1 2 2
c) 3 3 3 3
d) error

Answer: a
Explanation: The value of a[0] changes in each iteration. Since the first value that it takes is itself, there is no visible error in the current example.

---

112. What is the output of the following?
a = [0, 1, 2, 3]
i = -2
for i not in a:
print(i)
i += 1
a) -2 -1
b) 0
c) error
d) none of the mentioned

Answer: c
Explanation: SyntaxError, not in isn't allowed in for loops.

---

113. What is the output of the following?
string = 'my name is x'
for i in ' '.join(string.split()):
print (i, end=', ')
a) m, y, , n, a, m, e, , i, s, , x,
b) m, y, , n, a, m, e, , i, s, , x
c) my, name, is, x,
d) error

Answer: a
Explanation: Variable i takes the value of one character at a time.

---

114. What is the output of the following?
print('abc. DEF'.capitalize())
a) abc def
b) ABC DEF
c) Abc def
d) Abc Def

Answer: c
Explanation: The first letter of the string is converted to uppercase and the others are converted to lowercase.

---

115. What is the output of the following?
print('abcdef'.center())
a) cd
b) abcdef
c) error
d) none of the mentioned

Answer: c
Explanation: The function center() takes atleast one parameter.

---

116. What is the output of the following?
print('abcdef'.center(0))

a) cd
b) abcdef
c) error
d) none of the mentioned

Answer: c
Explanation: The entire string is printed when the argument passed to center() is less than the length of the string.

---

117. What is the output of the following?
print('*', 'abcdef'.center(7), '*')
a) * abcdef *
b) * abcdef *
c) *abcdef *
d) * abcdef*

Answer: b
Explanation: Padding is done towards the left-hand-side first when the final string is of odd length. Extra spaces are present since we haven't overridden the value of sep.

---

118. What is the output of the following?
print('*', 'abcdef'.center(7), '*', sep='')
a) * abcdef *
b) * abcdef *
c) *abcdef *
d) * abcdef*

Answer: d
Explanation: Padding is done towards the left-hand-side first when the final string is of odd length.

---

119. What is the output of the following?
print('*', 'abcde'.center(6), '*', sep='')
a) * abcde *
b) * abcde *
c) *abcde *
d) * abcde*

Answer: c
Explanation: Padding is done towards the right-hand-side first when the final string is of even length.

---

120. What is the output of the following?
print('abcdef'.center(7, 1))
a) 1abcdef
b) abcdef1
c) abcdef
d) error

Answer: d
Explanation: TypeError, the fill character must be a character, not an int.

---

121. What is the output of the following?
print('abcdef'.center(7, '1'))
a) 1abcdef
b) abcdef1
c) abcdef
d) error

Answer: a
Explanation: The character '1' is used for padding instead of a space.

---

122. What is the output of the following?
print('abcdef'.center(10, '12'))
a) 12abcdef12
b) abcdef1212
c) 1212abcdef
d) error

Answer: d
Explanation: The fill character must be exactly one character long.

123. What is the output of the following?
print('xyyzxyzxzxyy'.count('yy', 1))
a) 2
b) 0
c) 1
d) none of the mentioned

Answer: a
Explanation: Counts the number of times the substring ‘yy’ is present in the given string, starting from position 1.

124. What is the output of the following?
print('xyyzxyzxzxyy'.count('yy', 2))
a) 2
b) 0
c) 1
d) none of the mentioned

Answer: c
Explanation: Counts the number of times the substring ‘yy’ is present in the given string, starting from position 2.

125. What is the output of the following?
print('xyyzxyzxzxyy'.count('xyy', 0, 100))
a) 2
b) 0
c) 1
d) error

Answer: a
Explanation: An error will not occur if the end value is greater than the length of the string itself.

126. What is the output of the following?
print('xyyzxyzxzxyy'.count('xyy', 2, 11))
a) 2
b) 0
c) 1
d) error

Answer: b
Explanation: Counts the number of times the substring ‘xyy’ is present in the given string, starting from position 2 and ending at position 11.

127. What is the output of the following?
print('xyyzxyzxzxyy'.count('xyy', -10, -1))
a) 2
b) 0
c) 1
d) error

Answer: b
Explanation: Counts the number of times the substring ‘xyy’ is present in the given string, starting from position 2 and ending at position 11.

128. What is the output of the following?
print('abc'.encode())

a) abc
b) ‘abc’
c) b’abc’
d) h’abc’

Answer: c
Explanation: A bytes object is returned by encode.

---

129. What is the default value of encoding in encode()?
a) ascii
b) qwerty
c) utf-8
d) utf-16

Answer: c
Explanation: The default value of encoding is utf-8.

---

130. What is the output of the following?
print('xyyzxyzxzxyy'.endswith('xyy'))
a) 1
b) True
c) 3
d) 2

Answer: b
Explanation: The function returns True if the given string ends with the specified substring.

---

131. What is the output of the following?
print('xyyzxyzxzxyy'.endswith('xyy', 0, 2))
a) 0
b) 1
c) True
d) False

Answer: d
Explanation: The function returns False if the given string does not end with the specified substring.

---

132. What is the output of the following?
print('abcdef'.find('cd'))
a) True
b) 2
c) 3
d) none of the mentioned

Answer: b
Explanation: The first position in the given string at which the substring can be found is returned.

---

133. What is the output of the following?
print('ccdcddcd'.find('c'))
a) 4
b) 0
c) error
d) True

Answer: b
Explanation: The first position in the given string at which the substring can be found is returned.

---

134. What is the output of the following?
print('Hello {0} and {1}'.format('foo', 'bin'))
a) Hello foo and bin
b) Hello {0} and {1} foo bin
c) error
d) Hello 0 and 1

Answer: a
Explanation: The numbers 0 and 1 represent the position at which the strings are present.

---

135. What is the output of the following?
print('Hello {1} and {0}'.format('bin', 'foo'))

a) Hello foo and bin
b) Hello bin and foo
c) error
d) none of the mentioned

Answer: a
Explanation: The numbers 0 and 1 represent the position at which the strings are present.

---

136. What is the output of the following?
print('Hello {} and {}'.format('foo', 'bin'))
a) Hello foo and bin
b) Hello {} and {}
c) error
d) Hello and

Answer: a
Explanation: It is the same as Hello {0} and {1}.

---

137. What is the output of the following?
print('Hello {name1} and {name2}'.format('foo', 'bin'))
a) Hello foo and bin
b) Hello {name1} and {name2}
c) error
d) Hello and

Answer: c
Explanation: The arguments passed to the function format aren’t keyword arguments.

---

138. What is the output of the following?
print('Hello {0!r} and {0!s}'.format('foo', 'bin'))
a) Hello foo and bin
b) Hello ‘foo’ and bin
c) Hello foo and ‘bin’
d) error

Answer: b
Explanation: !r causes the charactes ‘ or ” to be printed as well.

---

139. What is the output of the following?
print('Hello {0} and {1}'.format(('foo', 'bin')))
a) Hello foo and bin
b) Hello (‘foo’, ‘bin’) and (‘foo’, ‘bin’)
c) error
d) none of the mentioned

Answer: c
Explanation: IndexError, the tuple index is out of range.

---

140. What is the output of the following?
print('Hello {0[0]} and {0[1]}'.format(('foo', 'bin')))

a) Hello foo and bin
b) Hello (‘foo’, ‘bin’) and (‘foo’, ‘bin’)
c) error
d) none of the mentioned

Answer: a
Explanation: The elements of the tuple are accessed by their indices.

---

141. What is the output of the following?
print('The sum of {0} and {1} is {2}'.format(2, 10, 12))
a) The sum of 2 and 10 is 12
b) error
c) The sum of 0 and 1 is 2
d) none of the mentioned

Answer: a
Explanation: The arguments passed to the function format can be integers also.

---

142. What is the output of the following?
print('The sum of {0:b} and {1:x} is {2:o}'.format(2, 10, 12))
a) The sum of 2 and 10 is 12
b) The sum of 10 and a is 14
c) The sum of 10 and a is c
d) error

Answer: b
Explanation: 2 is converted to binary, 10 to hexadecimal and 12 to octal.

---

143. What is the output of the following?
print('{:,}'.format(1112223334))
a) 1,112,223,334
b) 111,222,333,4
c) 1112223334
d) error

Answer: a
Explanation: A comma is added after every third digit from the right.

---

144. What is the output of the following?
print('{:,}'.format('1112223334'))
a) 1,112,223,334
b) 111,222,333,4
c) 1112223334
d) error

Answer: d
Explanation: An integer is expected.

---

145. What is the output of the following?
print('{:$}'.format(1112223334))
a) 1,112,223,334
b) 111,222,333,4
c) 1112223334
d) error

Answer: d
Explanation: $ is an invalid format code.

---

146. What is the output of the following?
print('{:#}'.format(1112223334))
a) 1,112,223,334
b) 111,222,333,4
c) 1112223334
d) error

Answer: c
Explanation: The number is printed as it is.

---

147. What is the output of the following?
print('{0:.2%}'.format(1/3))
a) 0.33
b) 0.33%
c) 33.33%
d) 33%

Answer: c
Explanation: The symbol % is used to represent the result of an expression as a percentage.

---

148. What is the output of the following?
print('ab12'.isalnum())
a) True
b) False
c) None
d) error

Answer: a
Explanation: The string has only letters and digits.

149. What is the output of the following?
print('ab,12'.isalnum())

a) True
b) False
c) None
d) error

Answer: b
Explanation: The character , is not a letter or a digit.

150. What is the output of the following?
print('ab'.isalpha())
a) True
b) False
c) None
d) error

Answer: a
Explanation: The string has only letters.

151. What is the output of the following?
print('a B'.isalpha())
a) True
b) False
c) None
d) error

Answer: b
Explanation: Space is not a letter.

152. What is the output of the following?
print('0xa'.isdigit())
a) True
b) False
c) None
d) error

Answer: b
Explanation: Hexadecimal digits aren’t considered as digits (a-f).

153. What is the output of the following?
print(''.isdigit())
a) True
b) False
c) None
d) error

Answer: b
Explanation: If there are no characters then False is returned.

154. What is the output of the following?
print('my_string'.isidentifier())
a) True
b) False
c) None
d) error

Answer: a
Explanation: It is a valid identifier.

---

155. What is the output of the following?
print('__foo__'.isidentifier())
a) True
b) False
c) None
d) error

Answer: a
Explanation: It is a valid identifier.

---

156. What is the output of the following?
print('abc'.islower())
a) True
b) False
c) None
d) error

Answer: a
Explanation: There are no uppercase letters.

---

157. What is the output of the following?
print('a@ 1,'.islower())
a) True
b) False
c) None
d) error

Answer: a
Explanation: There are no uppercase letters.

---

158. What is the output of the following?
print('11'.isnumeric())
a) True
b) False
c) None
d) error

Answer: a
Explanation: All the character are numeric.

---

159. What is the output of the following?
print('1.1'.isnumeric())
a) True
b) False
c) None
d) error

Answer: b
Explanation: The character . is not a numeric character.

---

160. What is the output of the following?
print('1@ a'.isprintable())
a) True
b) False
c) None
d) error

Answer: a
Explanation: All those characters are printable.

---

161. What is the output of the following?
print('''
'''.isspace())
a) True
b) False
c) None
d) error

Answer: a
Explanation: A newline character is considered as space.

---

162. What is the output of the following?
print('\t'.isspace())
a) True
b) False
c) None
d) error

Answer: a
Explanation: Tabspaces are considered as spaces.

---

163. What is the output of the following?
print('HelloWorld'.istitle())
a) True
b) False
c) None
d) error

Answer: b
Explanation: The letter W is uppercased.

---

164. What is the output of the following?
print('Hello World'.istitle())
a) True
b) False
c) None
d) Error

Answer: a
Explanation: It is in title form.

---

165. What is the output of the following?
print('1Rn@'.lower())
a) n
b) 1rn@
c) rn
d) r

Answer: b
Explanation: Uppercase letters are converted to lowercase. The other characters are left unchanged.

---

166. What is the output of the following?
print('''
\tfoo'''.lstrip())
a) \tfoo
b) foo
c) foo
d) none of the mentioned

Answer: b
Explanation: All leading whitespace is removed.

---

167. What is the output of the following?
print('xyyzxxyxyy'.lstrip('xyy'))

a) error
b) zxxyxyy
c) z
d) zxxy

Answer: b
Explanation: The leading characters containing xyy are removed.

---

168. What is the output of the following?
print('xyxxyyzxxy'.lstrip('xyy'))
a) zxxy
b) xyxxyyzxxy
c) xyxzxxy
d) none of the mentioned

Answer: a
Explanation: All combinations of the characters passed as an argument are removed from the left hand side.

---

169. What is the output of the following?
print('cba'.maketrans('abc', '123'))
a) {97: 49, 98: 50, 99: 51}
b) {65: 49, 66: 50, 67: 51}
c) 321
d) 123

Answer: a
Explanation: A translation table is returned by maketrans.

---

170. What is the output of the following?
print('a'.maketrans('ABC', '123'))
a) {97: 49, 98: 50, 99: 51}
b) {65: 49, 66: 50, 67: 51}
c) {97: 49}
d) 1

Answer: a
Explanation: maketrans() is a static method so it’s behaviour does not depend on the object from which it is being called.

---

171. What is the output of the following?
print('abcdef'.partition('cd'))
a) (‘ab’, ‘ef’)
b) (‘abef’)
c) (‘ab’, ‘cd’, ‘ef’)
d) 2

Answer: c
Explanation: The string is split into three parts by partition.

---

172. What is the output of the following?
print('abcdefcdgh'.partition('cd'))
a) (‘ab’, ‘cd’, ‘ef’, ‘cd’, ‘gh’)
b) (‘ab’, ‘cd’, ‘efcdgh’)
c) (‘abcdef’, ‘cd’, ‘gh’)
d) error

Answer: b
Explanation: The string is partitioned at the point where the separator first appears.

---

173. What is the output of the following?
print('abcd'.partition('cd'))
a) (‘ab’, ‘cd’, ”)
b) (‘ab’, ‘cd’)
c) error
d) none of the mntioned

Answer: a
Explanation: The last item is a null string.

---

174. What is the output of the following?
print('abef'.partition('cd'))
a) (‘abef’)
b) (‘abef’, ‘cd’, ”)
c) (‘abef’, ”, ”)
d) error

Answer: c
Explanation: The separator is not present in the string hence the second and the third elements of the tuple are null strings.

175. What is the output of the following?
print('abcdef12'.replace('cd', '12'))
a) ab12ef12
b) abcdef12
c) ab12efcd
d) none of the mentioned

Answer: a
Explanation: All occurences of the first substring are replaced by the second substring.

176. What is the output of the following?
print('abef'.replace('cd', '12'))
a) abef
b) 12
c) error
d) none of the mentioned

Answer: a
Explanation: The first substring is not present in the given string and hence nothing is replaced.

177. What is the output of the following?
print('abcefd'.replace('cd', '12'))
a) ab1ef2
b) abcefd
c) ab1efd
d) ab12ed2

Answer: b
Explanation: The first substring is not present in the given string and hence nothing is replaced.

178. What is the output of the following?
print('xyyxyyxyxyxxy'.replace('xy', '12', 0))
a) xyyxyyxyxyxxy
b) 12y12y1212x12
c) 12yxyyxyxyxxy
d) xyyxyyxyxyx12

Answer: a
Explanation: The first 0 occurances of the given substring are replaced.

179. What is the output of the following?
print('xyyxyyxyxyxxy'.replace('xy', '12', 100))
a) xyyxyyxyxyxxy
b) 12y12y1212x12
c) none of the mentioned
d) error

Answer: b
Explanation: The first 100 occurances of the given substring are replaced.

180. What is the output of the following?
print('abcdefcdghcd'.split('cd'))
a) [‘ab’, ‘ef’, ‘gh’]
b) [‘ab’, ‘ef’, ‘gh’, ”]
c) (‘ab’, ‘ef’, ‘gh’)
d) (‘ab’, ‘ef’, ‘gh’, ”)

Answer: b
Explanation: The given string is split and a list of substrings is returned.

---

181. What is the output of the following?
print('abcdefcdghcd'.split('cd', 0))
a) [‘abcdefcdghcd’]
b) ‘abcdefcdghcd’
c) error
d) none of the mentioned

Answer: a
Explanation: The given string is split at 0 occurances of the specified substring.

---

182. What is the output of the following?
print('abcdefcdghcd'.split('cd', -1))
a) [‘ab’, ‘ef’, ‘gh’]
b) [‘ab’, ‘ef’, ‘gh’, ”]
c) (‘ab’, ‘ef’, ‘gh’)
d) (‘ab’, ‘ef’, ‘gh’, ”)

Answer: b
Explanation: Calling the function with a negative value for maxsplit is the same as calling it without any maxsplit specified. The string will be split into as many substring s as possible.

---

183. What is the output of the following?
print('ab\ncd\nef'.splitlines())
a) [‘ab’, ‘cd’, ‘ef’]
b) [‘ab\n’, ‘cd\n’, ‘ef\n’]
c) [‘ab\n’, ‘cd\n’, ‘ef’]
d) [‘ab’, ‘cd’, ‘ef\n’]

Answer: a
Explanation: It is similar to calling split(‘\n’).

---

184. What is the output of the following?
print('Ab!2'.swapcase())
a) AB!@
b) ab12
c) aB!2
d) aB1@

Answer: c
Explanation: Lowercase letters are converted to uppercase and vice-versa.

---

185. What is the output of the following?
print('ab cd ef'.title())

a) Ab cd ef
b) Ab cd eF
c) Ab Cd Ef
d) none of the mentioned

Answer: c
Explanation: The first letter of every word is capitalized.

---

186. What is the output of the following?
print('ab cd-ef'.title())
a) Ab cd-ef
b) Ab Cd-ef
c) Ab Cd-Ef
d) none of the mentioned

Answer: c
Explanation: The first letter of every word is capitalized. Special symbols terminate a word.

---

187. What is the output of the following?
print('abcd'.translate('a'.maketrans('abc', 'bcd')))
a) bcde
b) abcd
c) error
d) none of the mentioned

Answer: d
Explanation: The output is bcdd since no translation is provided for d.

---

188. What is the output of the following?
print('abcd'.translate({97: 98, 98: 99, 99: 100}))
a) bcde
b) abcd
c) error
d) none of the mentioned

Answer: d
Explanation: The output is bcdd since no translation is provided for d.

---

189. What is the output of the following?
print('abcd'.translate({'a': '1', 'b': '2', 'c': '3', 'd': '4'}))
a) abcd
b) 1234
c) error
d) none of the mentioned

Answer: a
Explanation: The function translate expects a dictionary of integers. Use maketrans() instead of doing the above.

---

190. What is the output of the following?
print('ab'.zfill(5))
a) 000ab
b) 00ab0
c) 0ab00
d) ab000

Answer: a
Explanation: The string is padded with zeroes on the left hand side. It is useful for formatting numbers.

---

191. What is the output of the following?
print('+99'.zfill(5))
a) 00+99
b) 00099
c) +0099
d) +++99

Answer: c
Explanation: Zeroes are filled in between the first sign and the rest of the string.

---

192. Which is/are the basic I/O connections in file?
a) Standard Input
b) Standard Output
c) Standard Errors
d) All of the above
e) None of the above

Answer: d
Explanation: Standard input, standard output and standard error. Standard input is the data that goes to the program. The standard input comes from a keyboard. Standard output is where we print our data with the print keyword. Unless redirected, it is the terminal console. The standard error is a stream where programs write their error messages. It is usually the text terminal.

---

193. What is the output of this program?
import sysprint 'Enter your name: ', name = '' while True: c = sys.stdin.read(1) if c == '\n': break name = name + c print 'Your name is:', name
If entered name is sanfoundry
a) sanfoundry
b) sanfoundry, sanfoundry
c) San
d) None of the mentioned

Answer: a
Explanation: In order to work with standard I/O streams, we must import the sys module. The read() method reads one character from the standard input. In our example we get a promt saying “Enter your name”. We enter our name and press enter. The enter key generates the new line character: \n.

---

194. What is the output of this program?
import sys sys.stdout.write('
Hello\n')sys.stdout.write('Python\n')

a) Compilation Error
b) Runtime Error
c) Hello Python
d) Hello
Python

Answer: d
Explanation: None

---

195. Which of the following mode will refer to binary data?
a) r
b) w
c) +
d) b

Answer:d
Explanation: Mode Meaning is as explained below:

---

196. What is the pickling?
a) It is used for object serialization
b) It is used for object deserialization
c) None of the mentioned

Answer: a
Explanation: Pickle is the standard mechanism for object serialization. Pickle uses a simple stack-based virtual machine that records the instructions used to reconstruct the object. This makes pickle vulnerable to security risks by malformed or maliciously constructed data, that may cause the deserializer to import arbitrary modules and instantiate any object.

---

197. What is unpickling?
a) It is used for object serialization
b) It is used for object deserialization
c) None of the mentioned

Answer: b
Explanation: We have been working with simple textual data. What if we are working with objects rather than simple text? For such situations, we can use the pickle module. This module serializes Python objects. The Python objects are converted into byte streams and written to text files. This process is called pickling. The inverse operation, reading from a file and reconstructing objects is called deserializing or unpickling.

---

198. What is the correct syntax of open() function?
a) file = open(file_name [, access_mode][, buffering])
b) file object = open(file_name [, access_mode][, buffering])
c) file object = open(file_name)
d) None of the mentioned

Answer: b
Explanation: Open() function correct syntax with the parameter details as shown below:

---

199. Correct syntax of file.writelines() is?
a) file.writelines(sequence)
b) fileObject.writelines()
c) fileObject.writelines(sequence)
d) None of the mentioned

Answer: c
Explanation: The method writelines() writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. There is no return value.

---

200. Following is the syntax for writelines() method: fileObject.writelines(sequence). 10. Correct syntax of file.readlines() is?
a) fileObject.readlines(sizehint);
b) fileObject.readlines();
c) fileObject.readlines(sequence)
d) None of the mentioned

Answer: a
Explanation: The method readlines() reads until EOF using readline() and returns a list containing the lines. If the optional sizehint argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read.

---

201. In file handling, what does this terms means “r, a”?
a) read, append
b) append, read
c) All of the mentioned
d) None of the the mentioned

Answer: a
Explanation: r- reading, a-appending.

---

202. What is the use of “w” in file handling?
a) Read
b) Write
c) Append
d) None of the the mentioned

Answer: b
Explanation: This opens the file for writing. It will create the file if it doesn’t exist, and if it does, it will overwrite it.

---

203. What is the use of “a” in file handling?
a) Read
b) Write
c) Append
d) None of the the mentioned

Answer:c
Explanation: This opens the fhe file in appending mode. That means, it will be open for writing and everything will be written to the end of the file.

---

204. Which function is used to read all the characters?
a) Read()
b) Readcharacters()
c) Readall()
d) Readchar()

Answer:a
Explanation:The read function reads all characters fh = open(“filename”, “r”)
content = fh.read().

5. Which function is used to read single line from file?
a) Readline()
b) Readlines()
c) Readstatement()
d) Readfullline()

---

205. content = fh.readline().
6. Which function is used to write all the characters?
a) write()
b) writecharacters()
c) writeall()
d) writechar()

Answer: a
Explanation:To write a fixed sequence of characters to a file
fh = open(“hello.txt”,”w”)
write(“Hello World”).
7. Which function is used to write a list of string in a file
a) writeline()
b) writelines()
c) writestatement()
d) writefullline()

---

206. Whether we can create a text file in python?
a) Yes
b) No
c) None of the mentioned

Answer: a
Explanation: Yes we can create a file in python. Creation of file is as shown below.

---

207. file = open(“newfile.txt”, “w”)
file.write(“hello world in the new file\n”)
file.write(“and another line\n”)
file.close().

10. Which of the following is modes of both writing and reading in binary format in file.?
a) wb+
b) w
c) wb
d) w+

Answer: a
Explanation: Here is the description below

---

208. Which of the following is not a valid mode to open a file?
a) ab
b) rw
c) r+
d) w+

Answer: b
Explanation: Use r+, w+ or a+ to perform both read and write operations using a single file object.

---

209. What is the difference between r+ and w+ modes?
a) no difference
b) in r+ the pointer is initially placed at the beginning of the file and the pointer is at the end for w+
c) in w+ the pointer is initially placed at the beginning of the file and the pointer is at the end for r+
d) depends on the operating system

Answer: b
Explanation: none.

---

210. How do you get the name of a file from a file object (fp)?
a) fp.name
b) fp.file(name)
c) self.__name__(fp)
d) fp.__name__()

Answer: a
Explanation: name is an attribute of the file object.

---

211. Which of the following is not a valid attribute of a file object (fp)?
a) fp.name
b) fp.closed
c) fp.mode
d) fp.size

Answer: d
Explanation: fp.size has not been implemented.

---

212. How do you close a file object (fp)?
a) close(fp)
b) fclose(fp)
c) fp.close()
d) fp.__close__()

Answer: c
Explanation: close() is a method of the file object.

---

213. How do you get the current position within the file?
a) fp.seek()
b) fp.tell()
c) fp.loc
d) fp.pos

Answer: b
Explanation: It gives the current position as an offset from the start of file.

---

214. How do you rename a file?
a) fp.name = ‘new_name.txt’
b) os.rename(existing_name, new_name)
c) os.rename(fp, new_name)
d) os.set_name(existing_name, new_name)

Answer: b
Explanation: os.rename() is used to rename files.

---

215. How do you delete a file?
a) del(fp)
b) fp.delete()
c) os.remove(‘file’)
d) os.delete(‘file’)

Answer: c
Explanation: os.remove() is used to delete files.

---

216. How do you change the file position to an offset value from the start?
a) fp.seek(offset, 0)
b) fp.seek(offset, 1)
c) fp.seek(offset, 2)
d) none of the mentioned

Answer: a
Explanation: 0 indicates that the offset is with respect to the start.

---

217. What happens if no arguments are passed to the seek function?
a) file position is set to the start of file
b) file position is set to the end of file
c) file position remains unchanged
d) error

Answer: d
Explanation: seek() takes at least one argument.

---

218. Which of the following is a features of DocString?
a) Provide a convenient way of associating documentation with Python modules, functions, classes, and methods
b) All functions should have a docstring
c) Docstrings can be accessed by the __doc__ attribute on objects
d) All of the mentioned

Answer: d
Explanation: Python has a nifty feature called documentation strings, usually referred to by its shorter name docstrings. DocStrings are an important tool that you should make use of since it helps to document the program better and makes it easier to understand.

---

219. Which are the advantages of functions in python?
a) Reducing duplication of code
b) Decomposing complex problems into simpler pieces
c) Improving clarity of the code
d) Reuse of code
e) Information hiding
f) All of the mentioned

Answer: f
Explanation: None.

---

220. What are the two types of functions?
a) Custom function
b) Built-in function
c) User-Defined function
d) System function

Answer: b and c
Explanation: Built-in functions and user defined ones. The built-in functions are part of the Python language. Examples are: dir(), len() or abs(). The user defined functions are functions created with the def keyword.

---

221. Where is function defined?
a) Module
b) Class
c) Another function
d) None of the mentioned

Answer: a, b and c
Explanation: Functions can be defined inside a module, a class or another function.

---

222. What is called when a function is defined inside a class?
a) Module
b) Class
c) Another function
d) Method

Answer: d
Explanation: None.

---

223. Which of the following is the use of id() function in python?
a) Id returns the identity of the object
b) Every object doesn’t have a unique id
c) All of the mentioned
d) None of the mentioned

Answer: a
Explanation: Each object in Python has a unique id. The id() function returns the object’s id.

---

224. Which of the following refers to mathematical function?
a) sqrt
b) rhombus
c) add
d) rhombus

Answer: a
Explanation: Functions that are always available for usage, functions that are contained within external modules, which must be imported and functions defined by a programmer with the def keyword.

---

225. Eg: math import sqrt
The sqrt() function is imported from the math module.
7. What is the output of below program?
def cube(x): return x * x * x x = cube(3) print x

a) 9
b) 3
c) 27
d) 30

Answer: c
Explanation: A function is created to do a specific task. Often there is a result from such a task. The return keyword is used to return values from a function. A function may or may not return a value. If a function does not have a return keyword, it will send a none value.

---

226. What is the output of the below program?
def C2F(c): return c * 9/5 + 32print C2F(100)print C2F(0)
a) 212
32
b) 314
24
c) 567
98
d) None of the mentioned

Answer: a
Explanation: None.

---

227. What is the output of the below program?
def power(x, y=2): r = 1 for i in range(y): r = r * x return rprint power(3)print power(3, 3)

a) 212
32
b) 9
27
c) 567
98
d) None of the mentioned

Answer: b
Explanation: The arguments in Python functions may have implicit values. An implicit value is used, if no value is provided. Here we created a power function. The function has one argument with an implicit value. We can call the function with one or two arguments.

---

228. What is the output of the below program?
def sum(*args): '''Function returns the sum of all values''' r = 0 for i in args: r += i return rprint sum.__doc__print sum(1, 2, 3)print sum(1, 2, 3, 4, 5)
a) 6
15
b) 6
100
c) 123
12345
d) None of the mentioned

Answer: a
Explanation: We use the * operator to indicate, that the function will accept arbitrary number of arguments. The sum() function will return the sum of all arguments. The first string in the function body is called the function documentation string. It is used to document the function. The string must be in triple quotes.

---

229. Python supports the creation of anonymous functions at runtime, using a construct called _____?
a) Lambda
b) pi
c) anonymous
d) None of the mentioned

Answer: a
Python supports the creation of anonymous functions (i.e. functions that are not bound to a name) at runtime, using a construct called lambda. Lambda functions are restricted to a single expression. They can be used wherever normal functions can be used.
2. What is the output of this program?
y = 6 z = lambda x: x * yprint z(8)
a) 48
b) 14
c) 64
d) None of the mentioned

---

230. The lambda function is executed. The number 8 is passed to the anonymous function and it returns 48 as the result. Note that z is not a name for this function. It is only a variable to which the anonymous function was assigned.
3. What is the output of below program?
lamb = lambda x: x ** 3print(lamb(5))
a) 15
b) 555
c) 125
d) None of the mentioned

Answer: c
Explanation: None.

---

231. Is Lambda contains return statements
a) True
b) False

Answer: b
Explanation: lambda definition does not include a return statement — it always contains an expression which is returned. Also note that we can put a lambda definition anywhere a function is expected, and we don't have to assign it to a variable at all.

---

232. Lambda is a statement.
a) True
b) False

Answer: b
Explanation: None.

---

233. Lambda contains block of statements
a) True
b) False

Answer: b
Explanation: None.

---

234. What is the output of below program?
def f(x, y, z): return x + y + z f(2, 30, 400)
a) 432
b) 24000
c) 430
d) None of the mentioned

Answer: a
Explanation: None.

---

235. What is the output of below program?
def writer():title = 'Sir'name = (lambda x:title + ' ' + x)return name who = writer()who('Arthur')
a) Arthur Sir
b) Sir Arthur
c) Arthur
d) None of the mentioned

Answer: b
Explanation: None.

---

236. What is the output of this program?
L = [lambda x: x ** 2, lambda x: x ** 3, lambda x: x ** 4] for f in L:print(f(3))

a) 27
81
343
b) 6
9
12
c) 9
27
81
d) None of the mentioned

Answer: c
Explanation: None.

---

237. What is the output of this program?
min = (lambda x, y: x if x < y else y) min(101*99, 102*98)
a) 9997
b) 9999
c) 9996
d) None of the mentioned

Answer: c
Explanation: None.

---

238. How many except statements can a try-except block have?
a) zero
b) one
c) more than one
d) more than zero

Answer: d
Explanation: There has to be at least one except statement.

---

239. When will the else part of try-except-else be executed?
a) always
b) when an exception occurs
c) when no exception occurs
d) when an exception occurs in to except block

Answer: c
Explanation: The else part is executed when no exception occurs.

---

240. Is the following code valid?
try:
# Do something
except:
Do something
finally:
Do something
a) no, there is no such thing as finally
b) no, finally cannot be used with except
c) no, finally must come before except
d) yes

Answer: b
Explanation: Refer documentation.

---

241. Is the following code valid?
try:
# Do something
except:
# Do something
else:
# Do something

a) no, there is no such thing as else
b) no, else cannot be used with except
c) no, else must come before except
d) yes

Answer: d
Explanation: Refer documentation.

---

242. Can one block of except statements handle multiple exception?
a) yes, like except TypeError, SyntaxError [,…]
b) yes, like except [TypeError, SyntaxError]
c) no
d) none of the mentioned

Answer: a
Explanation: Each type of exception can be specified directly. There is no need to put it in a list.

---

243. When is the finally block executed?
a) when there is no exception
b) when there is an exception
c) only if some condition that has been specified is satisfied
d) always

Answer: d
Explanation: The finally block is always executed.

---

244. What is the output of the following code?
def foo():
try:
return 1
finally:
return 2
k = foo()
print(k)
a) 1
b) 2
c) 3
d) error, there is more than one return statement in a single try-finally block

Answer: b
Explanation: The finally block is executed even there is a return statement in the try block.

---

245. What is the output of the following code?
def foo():
try:
print(1)
finally:
print(2)
foo()

a) 1 2
b) 1
c) 2
d) none of the mentioned

Answer: a
Explanation: No error occurs in the try block so 1 is printed. Then the finally block is executed and 2 is printed.

---

246. What is the output of the following?
try:
if '1' != 1:
raise 'someError'
else:
print('someError has not occured')
except 'someError':
print ('someError has occured')
a) someError has occured
b) someError has not occured
c) invalid code
d) none of the mentioned

Answer: c
Explanation: A new exception class must inherit from a BaseException. There is no such inheritance here.

---

247. What happens when ‘1’ == 1 is executed?
a) we get a True
b) we get a False
c) an TypeError occurs
d) a ValueError occurs

Answer: b
Explanation: It simply evaluates to False and does not raise any exception.

---

248. What is the type of each element in sys.argv?
a) set
b) list
c) tuple
d) string

Answer: d
Explanation: It is a list of strings.

---

249. What is the length of sys.argv?
a) number of arguments
b) number of arguments + 1
c) number of arguments – 1
d) none of the mentioned

Answer: b
Explanation: The first argument is the name of the program itself. Therefore the length of sys.argv is one more than the number arguments.

---

250. What is the output of the following code?
def foo(k):
k[0] = 1
q = [0]
foo(q)
print(q)
a) [0]
b) [1]
c) [1, 0]
d) [0, 1]

Answer: b
Explanation: Lists are passed by reference.

---

251. How are keyword arguments specified in the function heading?
a) one star followed by a valid identifier
b) one underscore followed by a valid identifier
c) two stars followed by a valid identifier
d) two underscores followed by a valid identifier

Answer: c
Explanation: Refer documentation.

---

252. How many keyword arguments can be passed to a function in a single function call?
a) zero
b) one
c) zero or more
d) one or more

Answer: c
Explanation: zero keyword arguments may be passed if all the arguments have default values.

---

253. What is the output of the following code?
def foo(fname, val):
print(fname(val))
foo(max, [1, 2, 3])
foo(min, [1, 2, 3])
a) 3 1
b) 1 3
c) error
d) none of the mentioned

Answer: a
Explanation: It is possible to pass function names as arguments to other functions.

---

254. What is the output of the following code?
def foo():
return total + 1
total = 0
print(foo())
a) 0
b) 1
c) error
d) none of the mentioned

Answer: b
Explanation: It is possible to read the value of a global variable directly.

---

255. What is the output of the following code?
def foo():
total += 1
return total
total = 0
print(foo())
a) 0
b) 1
c) error
d) none of the mentioned

Answer: c
Explanation: It is not possible to change the value of a global variable without explicitly specifying it.

---

256. What is the output of the following code?
def foo(x):
x = ['def', 'abc']
return id(x)
q = ['abc', 'def']
print(id(q) == foo(q))
a) True
b) False
c) None
d) error

Answer: b
Explanation: A new object is created in the function.

---

257. What is the output of the following code?
def foo(i, x=[]):
x.append(i)
return x
for i in range(3):
print(foo(i))
a) [0] [1] [2]
b) [0] [0, 1] [0, 1, 2]
c) [1] [2] [3]
d) [1] [1, 2] [1, 2, 3]

Answer: b
Explanation: When a list is a default value, the same list will be reused.

---

258. How are variable length arguments specified in the function heading?
a) one star followed by a valid identifier
b) one underscore followed by a valid identifier
c) two stars followed by a valid identifier
d) two underscores followed by a valid identifier

Answer: a
Explanation: Refer documentation.

---

259. Which module in the python standard library parses options received from the command line?
a) getopt
b) os
c) getarg
d) main

Answer: a
Explanation: getopt parses options received from the command line.

---

260. What is the type of sys.argv?
a) set
b) list
c) tuple
d) string

Answer: b
Explanation: It is a list of elements.

---

261. What is the value stored in sys.argv[0]?
a) null
b) you cannot access it
c) the program’s name
d) the first argument

Answer: c
Explanation: Refer documentation.

---

262. How are default arguments specified in the function heading?
a) identifier followed by an equal to sign and the default value
b) identifier followed by the default value within backticks (“)
c) identifier followed by the default value within square brackets ([])
d) identifier

Answer: a
Explanation: Refer documentation.

---

263. How are required arguments specified in the function heading?
a) identifier followed by an equal to sign and the default value
b) identifier followed by the default value within backticks (“)
c) identifier followed by the default value within square brackets ([])
d) identifier

Answer: d
Explanation: Refer documentation.

---

264. What is the output of the following code?
def foo(x):
x[0] = ['def']
x[1] = ['abc']
return id(x)
q = ['abc', 'def']
print(id(q) == foo(q))
a) True
b) False
c) None
d) error

Answer: a
Explanation: The same object is modified in the function.

---

265. Where are the arguments recived from the command line stored?
a) sys.argv
b) os.argv
c) argv
d) none of the mentioned

Answer: a
Explanation: Refer documentation.

---

266. What is the output of the following?
def foo(i, x=[]):
x.append(x.append(i))
return x
for i in range(3):
y = foo(i)
print(y)
a) [[[0]], [[[0]], [1]], [[[0]], [[[0]], [1]], [2]]]
b) [[0], [[0], 1], [[0], [[0], 1], 2]]
c) [[0], None, [1], None, [2], None]
d) [[[0]], [[[0]], [1]], [[[0]], [[[0]], [1]], [2]]]

Answer: c
Explanation: append() returns None.

---

267. What is the output of print(k) in the following?
k = [print(i) for i in my_string if i not in 'aeiou']
print(k)
a) all characters of my_string that aren’t vowels
b) a list of Nones
c) list of Trues
d) list of Falses

Answer: b
Explanation: print() returns None.

---

268. What is the output of the following?
my_string = 'hello world'
k = [(i.upper(), len(i)) for i in my_string]
print(k)
a) [(‘HELLO’, 5), (‘WORLD’, 5)]
b) [(‘H’, 1), (‘E’, 1), (‘L’, 1), (‘L’, 1), (‘O’, 1), (‘ ‘, 1), (‘W’, 1), (‘O’, 1), (‘R’, 1), (‘L’, 1), (‘D’, 1)]
c) [(‘HELLO WORLD’, 11)]
d) none of the mentioned

Answer: b
Explanation: We are iterating over each letter in the string.

---

269. Which of the following is the correct expansion of list_1 = [expr(i) for i in list_0 if func(i)] ?
a)
list_1 = []
for i in list_0:
if func(i):
list_1.append(i)

b)
for i in list_0:
if func(i):
list_1.append(expr(i))
c)
list_1 = []
for i in list_0:
if func(i):
list_1.append(expr(i))
d) none of the mentioned

Answer: c
Explanation: We have to create an empty list, loop over the contents of the existing list and check if a condtion is satisfied before performing some operation and adding it to the new list.

---

270. What is the output of the following?
x = [i**+1 for i in range(3)]; print(x);
a) [0, 1, 2]
b) [1, 2, 5]
c) error, **+ is not a valid operator
d) error, ‘;’ is not allowed

Answer: a
Explanation: i**+1 is evaluated as (i)**(+1).

---

271. What is the output of the following?
print([i.lower() for i in 'HELLO'])
a) [‘h’, ‘e’, ‘l’, ‘l’, ‘o’]
b) ‘hello’
c) [‘hello’]
d) hello

Answer: a
Explanation: We are iterating over each letter in the string.

---

272. What is the output of the following?
print([i+j for i in 'abc' for j in 'def'])

a) [‘da’, ‘ea’, ‘fa’, ‘db’, ‘eb’, ‘fb’, ‘dc’, ‘ec’, ‘fc’]
b) [[‘ad’, ‘bd’, ‘cd’], [‘ae’, ‘be’, ‘ce’], [‘af’, ‘bf’, ‘cf’]]
c) [[‘da’, ‘db’, ‘dc’], [‘ea’, ‘eb’, ‘ec’], [‘fa’, ‘fb’, ‘fc’]]
d) [‘ad’, ‘ae’, ‘af’, ‘bd’, ‘be’, ‘bf’, ‘cd’, ‘ce’, ‘cf’]

Answer: d
Explanation: If it were to be executed as a nested for loop, i would be the outer loop and j the inner loop.

---

273. What is the output of the following?
print([[i+j for i in 'abc'] for j in 'def'])
a) [‘da’, ‘ea’, ‘fa’, ‘db’, ‘eb’, ‘fb’, ‘dc’, ‘ec’, ‘fc’]
b) [[‘ad’, ‘bd’, ‘cd’], [‘ae’, ‘be’, ‘ce’], [‘af’, ‘bf’, ‘cf’]]
c) [[‘da’, ‘db’, ‘dc’], [‘ea’, ‘eb’, ‘ec’], [‘fa’, ‘fb’, ‘fc’]]
d) [‘ad’, ‘ae’, ‘af’, ‘bd’, ‘be’, ‘bf’, ‘cd’, ‘ce’, ‘cf’]

Answer: b
Explanation: The inner list is generated once for each value of j.

---

274. What is the output of the following?
print([if i%2==0: i; else: i+1; for i in range(4)])
a) [0, 2, 2, 4]
b) [1, 1, 3, 3]
c) error
d) none of the mentioned

Answer: c
Explanation: Syntax error.

---

275. Which of the following is the same as list(map(lambda x: x**-1, [1, 2, 3]))?
a) [x**-1 for x in [(1, 2, 3)]]
b) [1/x for x in [(1, 2, 3)]]
c) [1/x for x in (1, 2, 3)]
d) error

Answer: c
Explanation: x**-1 is evaluated as (x)**(-1).

---

276. Which of the following will print True?

```
a = foo(2)
b = foo(3)
print(a < b)
```

a)

```
class foo:
def __init__(self, x):
self.x = x
def __lt__(self, other):
if self.x < other.x:
return False
else:
return True
```

b)

```
class foo:
def __init__(self, x):
self.x = x
def __less__(self, other):
if self.x > other.x:
return False
else:
return True
```

c)

```
class foo:
def __init__(self, x):
self.x = x
def __lt__(self, other):
if self.x < other.x:
return True
else:
return False
```

d)

```
class foo:
def __init__(self, x):
self.x = x
def __less__(self, other):
if self.x < other.x:
return False
else:
return True
```

Answer: c

Explanation: __lt__ overloads the < operator.

277. Which operator is overloaded by __lg__()?
a) <
b) >
c) !=
d) none of the mentioned

Answer: d
Explanation: __lg__() is invalid.

---

278. Which function overloads the >> operator?
a) __more__()
b) __gt__()
c) __ge__()
d) none of the mentioned

Answer: d
Explanation: __rshift__() overloads the >> operator.

---

279. Let A and B be objects of class Foo. Which functions are called when print(A + B) is executed?
a) __add__(), __str__()
b) __str__(), __add__()
c) __sum__(), __str__()
d) __str__(), __sum__()

Answer: a
Explanation: The function __add__() is called first since it is within the bracket. The function __str__() is then called on the object that we recieved after adding A and B.

---

280. Which operator is overloaded by the __or__() function?
a) ||
b) |
c) //
d) /

Answer: b
Explanation: The function __or__() overloads the bitwise OR operator |.

---

281. Which function overloads the // operator?
a) __div__()
b) __ceildiv__()
c) __floordiv__()
d) __truediv__()

Answer: c
Explanation: __floordiv__() is for //.

---

282. What the does random.seed(3) return?
a) True
b) None
c) 3
d) 1

Answer: b
Explanation: The function random.seed() always returns a None.

---

283. Which of the following cannot be returned by random.randrange(4)?
a) 0
b) 3
c) 2.3
d) none of the mentioned

Answer: c
Explanation: Only integers can be returned.

---

284. Which of the following is equivalent to random.randrange(3)?
a) range(3)
b) random.choice(range(0, 3))
c) random.shuffle(range(3))
d) random.select(range(3))

Answer: b
Explanation: It returns one number from the given range.

---

285. The function random.randint(4) can return only one of the following values. Which?
a) 4
b) 3.4
c) error
d) none of the mentioned

Answer: c
Explanation: Error, the function takes two arguments.

---

286. Which of the following is equivalent to random.randint(3, 6)?
a) random.choice([3, 6])
b) random.randrange(3, 6)
c) 3 + random.randrange(3)
d) 3 + random.randrange(4)

Answer: d
Explanation: random.randint(3, 6) can return any one of 3, 4, 5 and 6.

---

287. Which of the following will not be returned by random.choice(“1 ,”)?
a) 1
b) (space)
c) ,
d) none of the mentioned

Answer: d
Explanation: Any of the characters present in the string may be returned.

---

288. Which of the following will never be displayed on executing print(random.choice({0: 1, 2: 3}))?
a) 0
b) 1
c) KeyError: 1
d) none of the mentioned

Answer: a
Explanation: It will not print 0 but dict[0] i.e. 1 may be printed.

---

289. What does random.shuffle(x) do when x = [1, 2, 3]?
a) return a list in which the elements 1, 2 and 3 are in random positions
b) do nothing, it is a placeholder for a function that is yet to be implemented
c) shuffle the elements of the list in-place
d) none of the mentioned

Answer: c
Explanation: The elements of the list passed to it are shuffled in-place.

---

290. Which type of elements are accepted by random.shuffle()?
a) strings
b) lists
c) tuples
d) integers

Answer: b
Explanation: Strings and tuples are immutable and an integer has no len().

---

291. What is the range of values that random.random() can return?
a) [0.0, 1.0]
b) (0.0, 1.0]
c) (0.0, 1.0)
d) [0.0, 1.0)

Answer: d
Explanation: Any number that is greater than or equal to 0.0 and lesser than 1.0 can be returned.

---

292. What is returned by math.ceil(3.4)?
a) 3
b) 4
c) 4.0
d) 3.0

Answer: b
Explanation: The ceil function returns the smallest integer that is bigger than or equal to the number itself.

---

293. What is the value returned by math.floor(3.4)?
a) 3
b) 4
c) 4.0
d) 3.0

Answer: a
Explanation: The floor function returns the biggest number that is smaller than or equal to the number itself.

---

294. What is the output of print(math.copysign(3, -1))?
a) 1
b) 1.0
c) -3
d) -3.0

Answer: d
Explanation: The copysign function returns a float whose absolute value is that of the first argument and the sign is that of the second argument.

---

295. What is displayed on executing print(math.fabs(-3.4))?
a) -3.4
b) 3.4
c) 3
d) -3

Answer: b
Explanation: A negative floating point number is returned as a positive floating point number.

---

296. Is the function abs() same as math.fabs()?
a) sometimes
b) always
c) never
d) none of the mentioned

Answer: a
Explanation: math.fabs() always returns a float and does not work with complex numbers whereas the return type of abs() is determined by the type of value that is passed to it.

---

297. What is the value returned by math.fact(6)?
a) 720
b) 6
c) [1, 2, 3, 6]
d) error

Answer: d
Explanation: NameError, fact() is not defined.

---

298. What is the value of x if x = math.factorial(0)?
a) 0
b) 1
c) error
d) none of the mentioned

Answer: b
Explanation: Factorial of 0 is 1.

---

299. What is math.factorial(4.0)?
a) 24
b) 1
c) error
d) none of the mentioned

Answer: a
Explanation: The factorial of 4 is returned.

---

300. What is the output of print(math.factorial(4.5))?
a) 24
b) 120
c) error
d) none of the mentioned

Answer: c
Explanation: Factorial is only defined for non-negative integers.

---

301. What is math.floor(0o10)?
a) 8
b) 10
c) 0
d) 9

Answer: a
Explanation: 0o10 is 8 and floor(8) is 8.

---

302. What does the function math.frexp(x) return?
a) a tuple containing of the mantissa and the exponent of x
b) a list containing of the mantissa and the exponent of x
c) a tuple containing of the mantissa of x
d) a list containing of the exponent of x

Answer: a
Explanation: It returns a tuple with two elements. The first element is the mantissa and the second element is the exponent.

---

303. What is the result of math.fsum([.1 for i in range(20)])?
a) 2.0
b) 20
c) 2
d) 2.0000000000000004

Answer: a
Explanation: The function fsum returns an accurate floating point sum of the elements of its argument.

---

304. What is the result of sum([.1 for i in range(20)])?
a) 2.0
b) 20
c) 2
d) 2.0000000000000004

Answer: d
Explanation: There is some loss of accuracy when we use sum with floating point numbers. Hence the function fsum is preferable.

---

305. What is returned by math.isfinite(float(‘inf’))?
a) True
b) False
c) None
d) error

Answer: b
Explanation: float(‘inf’) is not a finite number.

---

306. What is returned by math.isfinite(float(‘nan’))?
a) True
b) False
c) None
d) error

Answer: b
Explanation: float(‘nan’) is not a finite number.

---

307. What is x if x = math.isfinite(float(‘0.0’))?
a) True
b) False
c) None
d) error

Answer: a
Explanation: float(‘0.0’) is a finite number.

---

308. What is the result of the following?
>>> -float('inf') + float('inf')
a) inf
b) nan
c) 0
d) 0.0

Answer: b
Explanation: The result of float(‘inf’)-float(‘inf’) is undefined.

309. What is the output of the following?
print(math.isinf(float('-inf')))

a) error, the minus sign shouldn’t havve been inside the brackets
b) error, there is no function called isinf
c) True
d) False

Answer: c
Explanation: -float(‘inf’) is the same as float(‘-inf’).

310. What is the value of x if x = math.ldexp(0.5, 1)?
a) 1
b) 2.0
c) 0.5
d) none of the mentioned

Answer: d
Explanation: The value returned by ldexp(x, y) is x * (2 ** y). In the current case x is 1.0.

311. What is returned by math.modf(1.0)?
a) (0.0, 1.0)
b) (1.0, 0.0)
c) (0.5, 1)
d) (0.5, 1.0)

Answer: a
Explanation: The first element is the fractional part and the second element is the integral part of the argument.

312. What is the result of math.trunc(3.1)?
a) 3.0
b) 3
c) 0.1
d) 1

Answer: b
Explanation: The integral part of the floating point number is returned.

313. What is the output of print(math.trunc(‘3.1’))?
a) 3
b) 3.0
c) error
d) none of the mentioned

Answer: c
Explanation: TypeError, a string does not have __trunc__ method.

314. Which of the following is the same as math.exp(p)?
a) e ** p
b) math.e ** p
c) p ** e
d) p ** math.e

Answer: b
Explanation: math.e is the constant defined in the math module.

---

315. What is returned by math.expm1(p)?
a) (math.e ** p) – 1
b) math.e ** (p – 1)
c) error
d) none of the mentioned

Answer: a
Explanation: One is subtracted from the result of math.exp(p) and returned.

---

316. What is the default base used when math.log(x) is found?
a) e
b) 10
c) 2
d) none of the mentioned

Answer: a
Explanation: The natural log of x is returned by default.

---

317. Which of the following aren’t defined in the math module?
a) log2()
b) log10()
c) logx()
d) none of the mentioned

Answer: c
Explanation: log2() and log10() are defined in the math module.

---

318. What is returned by int(math.pow(3, 2))?
a) 6
b) 9
c) error, third argument required
d) error, too many arguments

Answer: b
Explanation: math.pow(a, b) returns a ** b.

---

319. What is output of print(math.pow(3, 2))?
a) 9
b) 9.0
c) None
d) none of the mentioned

Answer: b
Explanation: math.pow() returns a floating point number.

---

320. What is the value of x if x = math.sqrt(4)?
a) 2
b) 2.0
c) (2, -2)
d) (2.0, -2.0)

Answer: b
Explanation: The function returns one floating point number.

---

321. What does math.sqrt(X, Y) do?
a) calculate the Xth root of Y
b) calculate the Yth root of X
c) error
d) return a tuple with the square root of X and Y

Answer: c
Explanation: The function takes only one argument.

---

322. What does os.name contain?
a) the name of the operating system dependent module imported
b) the address of the module os
c) error, it should’ve been os.name()
d) none of the mentioned

Answer: a
Explanation: It contains the name of the operating system dependent module imported such as ‘posix’, ‘java’ etc.

---

323. What does print(os.geteuid()) print?
a) the group id of the current process
b) the user id of the current process
c) both the group id and the user of the current process
d) none of the mentioned

Answer: b
Explanation: os.geteuid() gives the user id while the os.getegid() gives the group id.

---

324. What does os.getlogin() return?
a) name of the current user logged in
b) name of the superuser
c) gets a form to login as a different user
d) all of the above

Answer: a
Explanation: It returns the name of the user who is currently logged in and is running the script.

---

325. What does os.close(f) do?
a) terminate the process f
b) terminate the process f if f is not responding
c) close the file descriptor f
d) return an integer telling how close the file pointer is to the end of file

Answer: c
Explanation: When a file descriptor is passed as an argument to os.close() it will be closed.

---

326. What does os.fchmod(fd, mode) do?
a) change permission bits of the file
b) change permission bits of the directory
c) change permission bits of either the file or the directory
d) none of the mentioned

Answer: a
Explanation: The arguments to the function are a file descriptor and the new mode.

---

327. Which of the following functions can be used to read data from a file using a file descriptor?
a) os.reader()
b) os.read()
c) os.quick_read()
d) os.scan()

Answer: b
Explanation: None of the other functions exist.

---

328. Which of the following returns a string that represents the present working directory?
a) os.getcwd()
b) os.cwd()
c) os.getpwd()
d) os.pwd()

Answer: a
Explanation: The function getcwd() (get current working directory) returns a string that represents th present working directory.

---

329. What does os.link() do?
a) create a symbolic link
b) create a hard link
c) create a soft link
d) none of the mentioned

Answer: b
Explanation: os.link(source, destination) will create a hard link from source to destination.

---

330. Which of the following can be used to create a directory?
a) os.mkdir()
b) os.creat_dir()
c) os.create_dir()
d) os.make_dir()

Answer: a
Explanation: The function mkdir() creates a directory in the path specified.

---

331. Which of the following can be used to create a symbolic link?
a) os.symlink()
b) os.symb_link()
c) os.symblin()
d) os.ln()

Answer: a
Explanation: It is the function that allows you to create a symbolic link.

---

332. What is the output of the following?
print(999+1 is 1000)
print(1+1 is 2)

False, True
Python maintains a pool of objects representing the first few hundred integers and reuses them to save on memory and object creation. To make it even more confusing, the definition of what "small integer" is differs across Python versions.
A mitigation here is to never use the 'is' operator for value comparison. The is operator is designed to deal exclusively with object identities.

---

333. What is the output of the following?
print(2.2 * 3.0 == 3.3 * 2.0)

False
Explanation: The cause of the above phenomena is indeed a rounding error:
>>> (2.2 * 3.0).hex()
'0x1.a666666666667p+2'
>>> (3.3 * 2.0).hex()
'0x1.a666666666666p+2

---

334. What is the output of the following?
print(10**1000000 > float('infinity'))

False

---

335. What is the output of the following?
class X(object):
def __init__(self):
self.__private = 1
def get_private(self):
return self.__private
def has_private(self):
return hasattr(self, '__private')
x = X()
print(x.has_private())
print(x.get_private())

False, 1
Python does not support object attributes hiding. But there is a workaround based on the feature of double underscored attributes mangling. Although changes to attribute names occur only to code, attributes names hardcoded into string constants remain unmodified. This may lead to confusing behavior when a double underscored attribute visibly "hides" from getattr()/hasattr() functions.

---

336. What is the output of the following?
```
class X(object):
def __init__(self):
self.__private = 1
x = X()
print(x.__private)
x.__private = 2
print(x.__private)
print(hasattr(x, '__private'))
```

AttributeError: 'X' object has no attribute '__private'
2
True

---

337. What is Python?

Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++. It is also usable as an extension language for applications that need a programmable interface. Finally, Python is portable: it runs on many Unix variants, on the Mac, and on PCs under MS-DOS, Windows, Windows NT, and OS/2.

---

338. What are the rules for local and global variables in Python?

In Python, variables that are only referenced inside a function are implicitly global. If a variable is assigned a new value anywhere within the function's body, it's assumed to be a local. If a variable is ever assigned a new value inside the function, the variable is implicitly local, and you need to explicitly declare it as 'global'. Though a bit surprising at first, a moment's consideration explains this. On one hand, requiring global for assigned variables provides a bar against unintended side-effects. On the other hand, if global was required for all global references, you'd be using global all the time. You'd have to declare as global every reference to a builtin function or to a component of an imported module. This clutter would defeat the usefulness of the global declaration for identifying side-effects.

---

339. How do I copy an object in Python?

In general, try copy.copy() or copy.deepcopy() for the general case. Not all objects can be copied, but most can. Some objects can be copied more easily. Dictionaries have a copy() method:
newdict = olddict.copy()
Sequences can be copied by slicing: new_l = l[:]

---

340. How do I convert a number to a string?

To convert, e.g., the number 144 to the string '144', use the built-in function str(). If you want a hexadecimal or octal representation, use the built-in functions hex() or oct(). For fancy formatting, use the % operator on strings, e.g. "%04d" % 144 yields '0144' and "%.3f" % (1/3.0) yields '0.333'. (Instead of % better to use 'abc'.format() function). See the library reference manual for details.

---

341. How do I convert between tuples and lists?	the function tuple(seq) converts any sequence (actually, any iterable) into a tuple with the same items in the same order. For example, tuple([1, 2, 3]) yields (1, 2, 3) and tuple('abc') yields ('a', 'b', 'c'). If the argument is a tuple, it does not make a copy but returns the same object, so it is cheap to call tuple() when you aren't sure that an object is already a tuple. The function list(seq) converts any sequence or iterable into a list with the same items in the same order. For example, list((1, 2, 3)) yields [1, 2, 3] and list('abc') yields ['a', 'b', 'c']. If the argument is a list, it makes a copy just like seq[:] would.
342. What's a negative index?	Python sequences are indexed with positive numbers and negative numbers. For positive numbers 0 is the first index 1 is the second index and so forth. For negative indices -1 is the last index and -2 is the penultimate (next to last) index and so forth. Think of seq[-n] as the same as seq[len(seq)- n]. Using negative indices can be very convenient. For example S[:-1] is all of the string except for its last character, which is useful for removing the trailing newline from a string.
343. What is a class?	A class is the particular object type created by executing a class statement. Class objects are used as templates to create instance objects, which embody both the data (attributes) and code (methods) specific to a datatype. A class can be based on one or more other classes, called its base class(es). It then inherits the attributes and methods of its base classes. This allows an object model to be successively refined by inheritance. You might have a generic Mailbox class that provides basic accessor methods for a mailbox, and subclasses such as MboxMailbox, MaildirMailbox, OutlookMailbox that handle various specific mailbox formats.
344. How do I call a method defined in a base class from a derived class that overrides it?	Use the built-in super() function: class Derived(Base): def meth (self): super(Derived, self).meth()
345. Where is the math.py (socket.py, regex.py, etc.) source file?	There are (at least) three kinds of modules in Python: 1. modules written in Python (.py); 2. modules written in C and dynamically loaded (.dll, .pyd, .so, .sl, etc); 3. modules written in C and linked with the interpreter; to get a list of these, type: import sys print sys.builtin_module_names
346. What is self?	Self is merely a conventional name for the first argument of a method. A method defined as meth(self, a, b, c) should be called as x.meth(a, b, c) for some instance x of the class in which the definition occurs; the called method will think it is called as meth(x, a, b, c). opening position.

347. How do I apply a method to a sequence of objects?	Use a list comprehension: result = [obj.method() for obj in List] More generically, you can try the following function: def method_map(objects, method, arguments): """method_map([a,b], "meth", (1,2)) gives [a.meth(1,2), b.meth(1,2)]""" nobjects = len(objects) methods = map(getattr, objects, [method]*nobjects) return map(apply, methods, [arguments]*nobjects)
348. How can I execute arbitrary Python statements from C?	The highest-level function to do this is PyRun_SimpleString() which takes a single string argument to be executed in the context of the module __main__ and returns 0 for success and -1 when an exception occurred (including SyntaxError). If you want more control, use PyRun_String(); see the source for PyRun_SimpleString() in Python/pythonrun.c.
349. What is Freeze for Windows?	Freeze is a program that allows you to ship a Python program as a single stand-alone executable file. It is not a compiler; your programs don't run any faster, but they are more easily distributable, at least to platforms with the same OS and CPU.
350. How do I interface to C++ objects from Python?	Depending on your requirements, there are many approaches. To do this manually, begin by reading the "Extending and Embedding" document. Realize that for the Python run-time system, there isn't a whole lot of difference between C and C++ -- so the strategy of building a new Python type around a C structure (pointer) type will also work for C++ objects
351. How do I generate random numbers in Python?	The standard module random implements a random number generator. Usage is simple: import random random.random() - This returns a random floating point number in the range [0, 1)

352. What will be the output of the code below?
def extendList(val, list=[]):
list.append(val)
return list
list1 = extendList(10)
list2 = extendList(123,[])
list3 = extendList('a')
print "list1 = %s" % list1
print "list2 = %s" % list2
print "list3 = %s" % list3
How would you modify the definition of extendList to produce the presumably desired behavior?

list1 = [10, 'a']
list2 = [123]
list3 = [10, 'a']
Explanation: New default list is created only once when the function is defined, and that same list is then used subsequently whenever extendList is invoked without a list argument being specified. This is because expressions in default arguments are calculated when the function is defined, not when it's called.
list1 and list3 are therefore operating on the same default list, whereas list2 is operating on a separate list that it created (by passing its own empty list as the value for the list parameter).
def extendList(val, list=None):
if list is None:
list = []
list.append(val)
return list

---

353. What will be the output of the code below?

def multipliers():
return [lambda x : i * x for i in range(4)]
print([m(2) for m in multipliers()])
How would you modify the definition of multipliers to produce the presumably desired behavior?

[6, 6, 6, 6]
Explanation: The reason for this is that Python's closures are late binding. This means that the values of variables used in closures are looked up at the time the inner function is called. So as a result, when any of the functions returned by multipliers() are called, the value of i is looked up in the surrounding scope at that time. By then, regardless of which of the returned functions is called, the for loop has completed and i is left with its final value of 3. Therefore, every returned function multiplies the value it is passed by 3, so since a value of 2 is passed in the above code, they all return a value of 6 (i.e., 3 x 2).
def multipliers():
for i in range(4): yield lambda x : i * x
def multipliers():
return [lambda x, i=i : i * x for i in range(4)]
or
from functools import partial
from operator import mul
def multipliers():
return [partial(mul, i) for i in range(4)]

---

354. What will be the output of the code below?
```
class Parent(object):
x = 1
class Child1(Parent):
pass
class Child2(Parent):
pass
print(Parent.x, Child1.x, Child2.x)
Child1.x = 2
print(Parent.x, Child1.x, Child2.x)
Parent.x = 3
print(Parent.x, Child1.x, Child2.x)
```

Answer:
1 1 1
1 2 1
3 2 3
Explanation:
in Python, class variables are internally handled as dictionaries. If a variable name is not found in the dictionary of the current class, the class hierarchy (i.e., its parent classes) are searched until the referenced variable name is found (if the referenced variable name is not found in the class itself or anywhere in its hierarchy, an AttributeError occurs).
Therefore, setting x = 1 in the Parent class makes the class variable x (with a value of 1) referenceable in that class and any of its children. That's why the first print statement outputs 1 1 1.
Subsequently, if any of its child classes overrides that value (for example, when we execute the statement Child1.x = 2), then the value is changed in that child only. That's why the second print statement outputs 1 2 1.
Finally, if the value is then changed in the Parent (for example, when we execute the statement Parent.x = 3), that change is reflected also by any children that have not yet overridden the value (which in this case would be Child2). That's why the third print statement outputs 3 2 3.

---

355. What will be the output of the code below in Python 2?
```
def div1(x,y):
print "%s/%s = %s" % (x, y, x/y)
def div2(x,y):
print "%s//%s = %s" % (x, y, x//y)
div1(5,2)
div1(5.,2)
div2(5,2)
div2(5.,2.)
```
Also, how would the answer differ in Python 3 (assuming, of course, that the above print statements were converted to Python 3 syntax)?

5/2 = 2
5.0/2 = 2.5
5//2 = 2
5.0//2.0 = 2.0
Explanation: By default, Python 2 automatically performs integer arithmetic if both operands are integers. As a result, 5/2 yields 2, while 5./2 yields 2.5. Also note that the "double-slash" (//) operator will always perform integer division, regardless of the operand types. That's why 5.0//2.0 yields 2.0 even in Python 2.
(Py 3+)
5/2 = 2.5
5.0/2 = 2.5
5//2 = 2
5.0//2.0 = 2.0

---

356. What will be the output of the code below?
```
list = ['a', 'b', 'c', 'd', 'e']
print(list[10:])
```

[]

Explanation: Attempting to access a slice of a list at a starting index that exceeds the number of members in the list will not result in an IndexError and will simply return an empty list. What makes this a particularly nasty gotcha is that it can lead to bugs that are really hard to track down since no error is raised at runtime.

---

357. What will be the ouput of lines 2, 4, 6, and 8?
1. list = [[ ]] * 5
2. list # output?
3. list[0].append(10)
4. list # output?
5. list[1].append(20)
6. list # output?
7. list.append(30)
8. list # output?

Answer: [[], [], [], [], []]
[[10], [10], [10], [10], [10]]
[[10, 20], [10, 20], [10, 20], [10, 20], [10, 20]]
[[10, 20], [10, 20], [10, 20], [10, 20], [10, 20], 30]
Explanation:
The first line of output is presumably intuitive and easy to understand; i.e., list = [[ ]] * 5 simply creates a list of 5 lists. However, the key thing to understand here is that the statement list = [[ ]] * 5 does NOT create a list containing 5 distinct lists; rather, it creates a a list of 5 references to the same list. With this understanding, we can better understand the rest of the output. list[0].append(10) appends 10 to the first list. But since all 5 lists refer to the same list, the output is: [[10], [10], [10], [10], [10]].
Similarly, list[1].append(20) appends 20 to the second list. But again, since all 5 lists refer to the same list, the output is now: [[10, 20], [10, 20], [10, 20], [10, 20], [10, 20]].
In contrast, list.append(30) is appending an entirely new element to the “outer” list, which therefore yields the output: [[10, 20], [10, 20], [10, 20], [10, 20], [10, 20], 30].

---

358. Given a list of N numbers, use a single list comprehension to produce a new list that only contains those values that are:
(a) even numbers, and
(b) from elements in the original list that had even indices
For example, if list[2] contains a value that is even, that value should be included in the new list, since it is also at an even index (i.e., 2) in the original list. However, if list[3] contains an even number, that number should not be included in the new list since it is at an odd index (i.e., 3) in the original list.

[x for x in list[::2] if x%2 == 0]

---

359. Given the following subclass of dictionary:
class DefaultDict(dict):
def __missing__(self, key):
return []
Will the code below work? Why or why not?
d = DefaultDict()
d['florp'] = 127

Yes, it will work. With this implementation of the DefaultDict class, whenever a key is missing, the instance of the dictionary will automatically be instantiated with a list.

---

360. What is Python really? You can (and are encouraged) make comparisons to other technologies in your answer

Python is an interpreted language. That means that, unlike languages like C and its variants, Python does not need to be compiled before it is run. Other interpreted languages include PHP and Ruby.
Python is dynamically typed, this means that you don't need to state the types of variables when you declare them or anything like that. You can do things like x=111 and then x="I'm a string" without error.
Python is well suited to object orientated programming in that it allows the definition of classes along with composition and inheritance. Python does not have access specifiers (like C++'s public, private), the justification for this point is given as "we are all adults here".
In Python, functions are first-class objects. This means that they can be assigned to variables, returned from other functions and passed into functions. Classes are also first class objects.
Writing Python code is quick but running it is often slower than compiled languages. Fortunately. Python allows the inclusion of C based extensions so bottlenecks can be optimised away and often are. The numpy package is a good example of this, it's really quite quick because a lot of the number crunching it does isn't actually done by Python.
Python finds use in many spheres - web applications, automation, scientific modelling, big data applications and many more. It's also often used as "glue" code to get other languages and components to play nice.
Python makes difficult things easy so programmers can focus on overriding algorithms and structures rather than nitty-gritty low level details.

---

361. Fill in the missing code:
def print_directory_contents(sPath):
"""
This function takes the name of a directory
and prints out the paths files within that
directory as well as any files contained in
contained directories.
This function is similar to os.walk. Please don't
use os.walk in your answer. We are interested inyour
ability to work with nested structures.
"""
fill_this_in

def print_directory_contents(sPath):
import os
for sChild in os.listdir(sPath):
sChildPath = os.path.join(sPath,sChild)
if os.path.isdir(sChildPath):
print_directory_contents(sChildPath)
else:
print(sChildPath)

---

362. Looking at the below code, write down the final values of A0, A1, ...An.
A0 = dict(zip(('a','b','c','d','e'),(1,2,3,4,5)))
A1 = range(10)
A2 = sorted([i for i in A1 if i in A0])
A3 = sorted([A0[s] for s in A0])
A4 = [i for i in A1 if i in A3]
A5 = {i:i*i for i in A1}
A6 = [[i,i*i] for i in A1]

Answer:
A0 = {'a': 1, 'c': 3, 'b': 2, 'e': 5, 'd': 4} # the order may vary
A1 = range(0, 10) # or [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] in python 2
A2 = []
A3 = [1, 3, 2, 5, 4]
A4 = [1, 2, 3, 4, 5]
A5 = {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
A6 = [[0, 0], [1, 1], [2, 4], [3, 9], [4, 16], [5, 25], [6, 36], [7, 49], [8, 64],

---

363. Python and multi-threading. Is it a good idea? List some ways to get some Python code to run in a parallel way.

Python doesn't allow multi-threading in the truest sense of the word. It has a multi-threading package but if you want to multi-thread to speed your code up, then it's usually not a good idea to use it. Python has a construct called the Global Interpreter Lock (GIL). The GIL makes sure that only one of your 'threads' can execute at any one time. A thread acquires the GIL, does a little work, then passes the GIL onto the next thread. This happens very quickly so to the human eye it may seem like your threads are executing in parallel, but they are really just taking turns using the same CPU core. All this GIL passing adds overhead to execution. This means that if you want to make your code run faster then using the threading package often isn't a good idea.

There are reasons to use Python's threading package. If you want to run some things simultaneously, and efficiency is not a concern, then it's totally fine and convenient. Or if you are running code that needs to wait for something (like some IO) then it could make a lot of sense. But the threading library wont let you use extra CPU cores.
Multi-threading can be outsourced to the operating system (by doing multi-processing), some external application that calls your Python code (eg, Spark or Hadoop), or some code that your Python code calls (eg: you could have your Python code call a C function that does the expensive multi-threaded stuff).

---

364. What does this code output:
def f(x,l=[]):
for i in range(x):
l.append(i*i)
print(l)
f(2)
f(3,[3,2,1])
f(3)

[0, 1]
[3, 2, 1, 0, 1, 4]
[0, 1, 0, 1, 4]

---

365. What is monkey patching and is it ever a good idea?

Monkey patching is changing the behaviour of a function or object after it has already been defined. For example:
import datetime
datetime.datetime.now=lambda:datetime.datetime(2012, 12, 12)
Most of the time it's a pretty terrible idea - it is usually best if things act in a well-defined way. One reason to monkey patch would be in testing. The mock package is very useful to this end.

---

366. What does this stuff mean: *args, **kwargs? And why would we use it?

Use *args when we aren't sure how many arguments are going to be passed to a function, or if we want to pass a stored list or tuple of arguments to a function. **kwargs is used when we dont know how many keyword arguments will be passed to a function, or it can be used to pass the values of a dictionary as keyword arguments. The identifiers args and kwargs are a convention, you could also use *bob and **billy but that would not be wise.
Here is a little illustration:
def f(*args,**kwargs):
print(args, kwargs)
l = [1,2,3]
t = (4,5,6)
d = {'a':7,'b':8,'c':9}
f()
f(1,2,3) # (1, 2, 3) {}
f(1,2,3,"groovy") # (1, 2, 3, 'groovy') {}
f(a=1,b=2,c=3) # () {'a': 1, 'c': 3, 'b': 2}
f(a=1,b=2,c=3,zzz="hi") # () {'a': 1, 'c': 3, 'b': 2, 'zzz': 'hi'}
f(1,2,3,a=1,b=2,c=3) # (1, 2, 3) {'a': 1, 'c': 3, 'b': 2}
f(*l,**d) # (1, 2, 3) {'a': 7, 'c': 9, 'b': 8}
f(*t,**d) # (4, 5, 6) {'a': 7, 'c': 9, 'b': 8}
f(1,2,*t) # (1, 2, 4, 5, 6) {}
f(q="winning",**d) # () {'a': 7, 'q': 'winning', 'c': 9, 'b': 8}
f(1,2,*t,q="winning",**d) # (1, 2, 4, 5, 6) {'a': 7, 'q': 'winning', 'c': 9, 'b': 8}

---

367. What do these mean to you: @classmethod, @staticmethod, @property?

Answer:

These are decorators. A decorator is a special kind of function (or class: "property" is a class) that either takes a function and returns a function, or takes a class and returns a class. The @ symbol is just syntactic sugar that allows you to decorate something in a way that's easy to read.
@my_decorator
def my_func(stuff):
do_things
Is equivalent to
def my_func(stuff):
do_things
my_func = my_decorator(my_func)

---

368. Describe Python's garbage collection mechanism in brief.

Python maintains a count of the number of references to each object in memory. If a reference count goes to zero then the associated object is no longer live and the memory allocated to that object can be freed up for something else
occasionally things called "reference cycles" happen. The garbage collector periodically looks for these and cleans them up. An example would be if you have two objects o1 and o2 such that o1.x == o2 and o2.x == o1. If o1 and o2 are not referenced by anything else then they shouldn't be live. But each of them has a reference count of 1.
Certain heuristics are used to speed up garbage collection. For example, recently created objects are more likely to be dead. As objects are created, the garbage collector assigns them to generations. Each object gets one generation, and younger generations are dealt with first.
This explanation is CPython specific.

---

369. Place the following functions below in order of their efficiency. They all take in a list of numbers between 0 and 1. The list can be quite long. An example input list would be [random.random() for i in range(100000)]. How would you prove that your answer is correct?
def f1(lIn):
l1 = sorted(lIn)
l2 = [i for i in l1 if i<0.5]
return [i*i for i in l2]
def f2(lIn):
l1 = [i for i in lIn if i<0.5]
l2 = sorted(l1)
return [i*i for i in l2]
def f3(lIn):
l1 = [i*i for i in lIn]
l2 = sorted(l1)
return [i for i in l1 if i<(0.5*0.5)]

Most to least efficient: f2, f1, f3. To prove that this is the case, you would want to profile your code. Python has a lovely profiling package that should do the trick.
import cProfile
lIn = [random.random() for i in range(100000)]
cProfile.run('f1(lIn)')
cProfile.run('f2(lIn)')
cProfile.run('f3(lIn)')

---

370. We have the following code with unknown function f(). In f(), we do not want to use return, instead, we may want to use generator.
for x in f(5):
print(x)
output:
0 1 8 27 64

def f(n):
for x in range(n):
yield x**3

---

371. What is __init__.py?

It is used to import a module in a directory, which is called package import. If we have a module, dir1/dir2/mod.py, we put __init__.py in each directories so that we can import the mod like this:
import dir1.dir2.mod
The __init__.py is usually an empty py file. The hierarchy gives us a convenient way of organizing the files in a large system.

372. Build a string with the numbers from 0 to 100, "0123456789101112..."

''.join([str(x) for x in range(101)])

373. Basic file processing: Printing contents of a file.

try:
with open('filename','r') as f:
print(f.read())
except IOError:
print("No such file exists")

374. What are container sequences and flat sequences?

Container sequences hold references to the objects they contain, which may be of any type, while flat sequences physically store the value of each item within its own memory space, and not as distinct object.

containers seq: list, tuple, collections.deque
flat seq: str, bytes, bytearray, memoryview and array.array

375. Is it better to build class with attributes or namedtuple?

Instances of class that you build with namedtuple take exactly the same amount of memory as tuples because the filed names are stored in the class. They use less memory than regular object because they don't store attributes in per a per-instance __dict__.

376. What will be the output?
>>> l = list(range(10))
>>> l[2:5] = [20, 30]
>>> del l[5:7]
>>> l

[0, 1, 20, 30, 5, 8, 9]

377. What will be the output?
>>> l = list(range(20))
>>> my_slice = slice(0, None, 2)
>>> l[my_slice]

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

378. What will be the output?
>>> head, *body, tail = range(10)
>>> head, body, tail

(0, [1, 2, 3, 4, 5, 6, 7, 8], 9)

379. What will be the output?

```
>>> weird_board = [['_'] *3] *3
>>> weird_board[1][2] = 'xD'
>>> weird_board
```

[['_', '_', 'xD'], ['_', '_', 'xD'], ['_', '_', 'xD']]

---

380. What will be the output?

```
>>> t = (1, 2, [30, 40])
>>> t[2] += [50, 60]
>>> t
```

After line:
>>> t[2] += [50, 60]
Traceback (most recent call last):
File "<input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
but:
>>> t
(1, 2, [30, 40, 50, 60])

---

381. Which sort algorithm is used in sorted() and list.sort()?

The sorting algorithm used in sorted() and list.sort() is Timsort, an adaptive algorithm that switches from insertion sort to merge sort strategies, depended on how ordered the data is.

---

382. What is hashable?

An object is hashable if it has a hash value which never changes during its lifetime (it needs a __hash__ metod), and can be compared to other objects (it needs an __eq__ method). Hashable objects which compare equal must have the same has value.
The atomic immutable types (str, bytes, numeric types) are all hashable.
Tuple is hashable only if all its items are hashable.
User-defined types are hashable by default because their hash value is their id() and they all compare not equal. If an object implements __eq__ that takes into account its internal state, it may be hashable only if all its attributes are immutable.

---

383. Can you simplify this code?

```
if key not in my_dict:
my_dict[key] = []
my_dict[key].append(new_value)
```

my_dict.setdefault(key, []).append(new_value)

(now only one lookup!)

---

384. Can you describe how retrieving an item from a dict works?

1. Calcuate hash from key
2. Use part of hash to locate a bucket in hash table.
3. Empty bucket?
yes - raise KeyError
no - 4.
4. Equal keys?
no - return value from bucket
yes - 5.
5. (hash collision) Use other parts of hash to locate a different hash table row and go to 3.

---

385. Why adding items to a dict may change the order of existing keys?

Whenever you add a new item to a dict, the Python interpreter may decide that hash table of that dictionary needs to grow. This entails building a new, bigger hash table, and adding all current items to the new table. During this process, new (but different) hash collisions may happen, with the result that the keys are likely to be ordered differently in the new hash table.

If you are iterating over the dictionary keys and changing them at the same time, your loop may not scan all the items as expected.

---

386. What is first-class object?

A program entity that can be:
- created at runtime
- assigned to variable or element in a data structure
- passed as an argument to a function
- returned as the result of a function

---

387. How would you execute same method on different objects?

```
from operator import methodcaler
text = 'xdddd'
upercase = methodcaller('upper')

upercase(text)

>>> 'XDDDD'

hiphenate = methodcaller('replace', 'x', 'X')
hiphenate(text)

>>> 'Xdddd'
```

---

388. What is and does functools.partial?

It's higher-order function that allows partial application of a function. Given a function, a partial application produces a new callable with some of the arguments of the orginal function fixed.

```
>>> triple = partial(mul, 3)
>>> triple(7)
21
```

---

389. What is decorator?

A decorator is a callable that takes another function (or class) as argument. The decorator may perform some processing with given function, and returns it or replace it with another function or callable object.

@decorate
def target():
print('xD')
is same as:

def target():
print('xD')
target = decorate(target)

Decorators run right after the decorated function is defined - usually its import time.

---

390. What will be the output?

```
>>> b = 6
>>> def p(a):
>>> print(a)
>>> print(b)
>>> p(3) #1 ?
```

and

```
>>> b = 6

>>> def p(a):
>>> print(a)
>>> print(b)
>>> b = 4
>>> p(3) #2 ?
```

1)
3
6
2)
3
error `UnboundLocalError: local variable 'b' referenced before assignment`

---

391. What is closure?

Closure is a function with an extended scope that encompasses nonglobal variables referenced in the body of the function but not defined there. It can access nonglobal variables that are defined outside of its body.

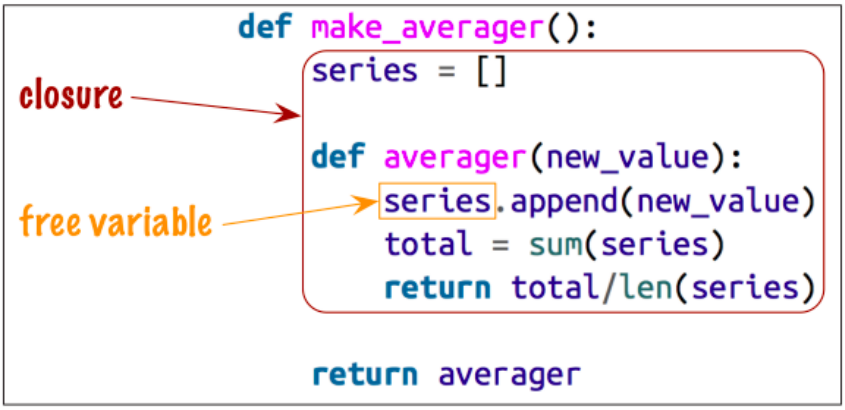

Figure 7-1. The closure for averager extends the scope of that function to include the binding for the free variable series.

Free variable is a technical term meaning a variable that is not bound in the local scope. (func_name.__code__.co_freevars)

---

392. What is nonlocal?

Introduced in Py3.4. Let's you flag a variable as a free variable even it is assigned a new value within the function. If a new value is assigned to nonlocal variable, the binding stored in the closure is changed.

---

393. What is normal notation for stacked decorators? (without syntax sugar)

@foo
@boo
def spam():
print('xD')

spam = foo(boo(spam))

---

394. How to create parameterized decorators?

Make a decorator factory that takes arguments and returns a decorator, which is then applied to the function to be decorated.

@deco(arg_1=True)
def foo():
print('xD')

foo = deco(arg_1=True)(foo)

---

395. Why do we use functools.wraps?

Normal decorator which returns new function masks __name__ and __doc__ of the decorated function. functools.wraps fixes that.

---

396. Whats the difference between == and 'is'?

The == operator compares the values of objects (data they hold). a == b is equal to a.__eq__(b).
'is' compares their identities.

The 'is' operator is faster than ==, because it cannot be overloaded, so Python does not have to find and invoke special methods to evaluate it.

---

397. What does 'del'?

The 'del' statement deletes names, not objects. An object can may be garbage collected as result of a 'del' command, but only if the variable deleted holds the last reference to the object, or if the object becomes unreachable (two objects refer to each other case)

---

398. What is the difference between repr() and str()?

repr() - return a string representing the object as the developer wants to see it.
str() - return a string representing the object as the user wants to see it.

---

399. classmethod vs staticmethod?

Decorators:
classmethod - changes the way the method is called, so it recives the class itself as the first argument, instead of instance. Its most common use for alternative constructors.
staticmethod - method recives no special first argument. In essence static method is just like a plain function that happens to live in a class body.

---

400. private and "protected" attributes in Python?

If you name an arribute with two leading underscores like '__x' Python stores the name in the instance __dict__ prefixed with leading underscore and the class name. So it will looks like '_ClassName__x'. This language feature is called name mangling.

Name mangling is about safety, not security: prevent accidental access. It still possible to get that value by calling `instance._ClassName__x`.

Single _ prefix has no special meaning to Python interpreter. It's just a convention to not use such a attribite outside a class.

---

401. What does __slots__ do?

Let interpreter store the instance attributes in a tuple instead of dict.

class Foo:
__slots__ = ('x', 'y')
...

by definig slots in class you are telling interpreter "these are all instance attributes in this class"

---

402. __getattr__ vs __getattribute__?

A key difference between __getattr__ and __getattribute__ is that __getattr__ is only invoked if the attribute wasn't found the usual ways. It's good for implementing a fallback for missing attributes. __getattribute__ allways called when there is an attempt to retrive the named attribute, except when the attribute is a special attribute or method. Dot notation and getattr and hasattr built-ins trigger this method. __getattr__ is only invoked after __getattribute__, and only when __getattribute__ raises AttributeError. To retrive attributes of the instance obj without triggering an infinite recursion, implementations of __getattribute__ should use super().__getattribute__(obj, name)

---

403. What is duck typing?

A programming style which does not look at an object’s type to determine if it has the right interface; instead, the method or attribute is simply called or used (“If it looks like a duck and quacks like a duck, it must be a duck.”) By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using type() or isinstance(). (Note, however, that duck-typing can be complemented with abstract base classes.) Instead, it typically employs hasattr() tests or EAFP programming.

---

404. What means EAFP?

Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many try and except statements. The technique contrasts with the LBYL style common to many other languages such as C.

---

405. Differences between Py2 and Py3?

- print function...
- Integer division:
>>> print '3 / 2 =', 3 / 2
3 / 2 = 1
>>> print('3 / 2 =', 3 / 2)
3 / 2 = 1.5
- For-loop variables and the global namespace leak...
- Returning iterable objects instead of lists
...

---

406. What is super()?

super() lets you avoid referring to the base class explicitly.
Note that the syntax changed in Python 3.0: you can just say super().__init__() instead of super(ChildB, self).__init__() which IMO is quite a bit nicer.

---

407. Built-in collections and objects?

mutable:
list, set, dict, bytearray
immutable:
tuple, frozenset
int, float, bool, string, unicode

---

408. Decimal vs float?

The decimal module implements fixed and floating point arithmetic using the model familiar to most people, rather than the IEEE floating point version implemented by most computer hardware. A Decimal instance can represent any number exactly, round up or down, and apply a limit to the number of significant digits.

---

409. What is lambda?

In Python, anonymous function means that a function is without a name. As we already know that def keyword is used to define the normal functions and the lambda keyword is used to create anonymous functions.
It has the following syntax:
lambda arguments: expression
This function can have any number of arguments but only one expression, which is evaluated and returned.
One is free to use lambda functions wherever function objects are required.
You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.
It has various uses in particular fields of programming besides other types of expressions in functions.
Lambda functions can be used along with built-in functions like filter(), map() and reduce().

---

410. What is MRO?

Method Resolution Order. Classes have an attribute called __mro__ holding a tuple of references to the superclasses in MRO order, from current class all the way to the object class.
class A:
def foo(self):
print('foo AAA')
class B(A):
def boo(self):
print('boo BBB')
class C(A):
def boo(self):
print('boo CCC')
def foo(self):
super().foo()
print('foo CCC')
class D(B, C):
pass
d = D()
d.foo()
d.boo()
print(D.__mro__)
output:
foo AAA
foo CCC
boo BBB
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>)

---

411. What is the output?

>>> x = 100
>>> ~x

-101

~x == -(x+1)

---

412. NotImplemented vs NotImplementedError?

NotImplemented is a special singleton value that an infix operator special method should return to tell the interpreter it cannot handle a given operand.
NotImplementedError is an exception that stub methods in abstract classes raise to warn that they must be overwritten by subclasses.

---

413. What is iterable?

Any object from which the iter built-in function can obtain an iterator. Objects implementing an __iter__ method returning an iterator are iterable. Sequences are always iterable; as are objects implementing __getitem__ method that takes 0-based indexes. (if __iter__ is not implemented, but __getitem__ is, Python creates an iterator that attempts to fetch items in order, starting from index 0)

Whenever the interpreter needs to iterate over an object x, it automatically calls iter(x).

An iterable should never act as an iterator over itself. Must implement __iter__, but not __next__.

```
class Foo:
def __init__(self, iterable):
self.x = iterable
def __iter__(self):
print('Creating an iterator!')
return iter(self.x)
```

414. What is Iterator?

Any object that implements the __next__ no-argument method that returns the next item in a series or raise StopIteration when there are no more items.
Python iterators also implement the __iter__ (has to return self) method so they are iterable as well. (to check if object is an iterator - form docs - use hasattr for both __iter__ and __next__ or - which is equal (cause of __subclasshook__) use isinstance(x, abc.Iterator))

Iterator should always be iterable, An iterator's __iter__ should just return self.

```
class Dupa:
def __init__(self, iterable):
self.x = iterable
def __iter__(self):
print('Creating iterator!')
return DupaIterator(self.x)
class DupaIterator:
def __init__(self, data):
self.data = data
self.index = 0
def __next__(self):
try:
d = self.data[self.index] + '_i'
except IndexError:
raise StopIteration
self.index += 1
return d
def __iter__(self):
return self
dupa = Dupa('dupa')
for i in dupa:
print(i)
```

output:

```
Creating iterator!
d_i
u_i
p_i
a_i
```

415. What is generator?

Any python function that has yield keyword in it's body is a generator function; a function which, when called, returns a generator object. Generator function is a generator factory.
def my_gen():
yield 'Elo'
for i in range(10):
yield 'ziomek_'+str(i)
return 'Elo 2'
To get returned value:

while True:
try:
next(g)
except StopIteration as e:
print(e.value) # 'Elo 2'
break

---

416. What does 'else' block do in for, while and try?

for - else block will run only if and when the for loop runs to completion (i.e not if loop is aborted with break)
while - else block will run only if and when the while loop exits becasue the condition became falsy (i.e not if loop is aborted with break)
try - else block will run only if no exception is raised in the try block.
In all cases, the else claue is also skipped if an exception or a return, break, or continue statement causes control jump out of the main block.

---

417. What is LBYL?

Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the EAFP approach and is characterized by the presence of many if statements.
In a multi-threaded environment, the LBYL approach can risk introducing a race condition between “the looking” and “the leaping”. For example, the code, if key in mapping: return mapping[key] can fail if another thread removes key from mapping after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

---

418. What is mixin?

A mixin is a special kind of multiple inheritance. There are two main situations where mixins are used:
- You want to provide a lot of optional features for a class.
- You want to use one particular feature in a lot of different classes.
class Request(AuthenticationMixin, UserAgentMixin, BaseRequest):
pass
Mixins are a sort of class that is used to "mix in" extra properties and methods into a class. This allows you to create classes in a compositional style.

---

419. What is context manager?

Context managers exist to control with statement. Context manager protocol consists of the __enter__ and __exit__ methods.
class MyContext:
def __enter__(self):
return 'xDDDD' # "as" object
def __exit__(self, exc_type, exc_val, exc_tb):
return True # says that Exception was handled, anything else will propagate Exception
shorter version:
@contextlib.contextmanager
def mycontext():
# __enter__ block code
yield 'xDDDD'
__exit__ block code
if exception was passed gen.throw(excepton) is invoked, causing th exception to be raised in the yield line.

---

420. What is coroutine briefly?

In Python, coroutines are similar to generators but with few extra methods and slight change in how we use yield statement. Generators produce data for iteration while coroutines can also consume data.
def grep(pattern):
pattern = pattern.lower()
print('Started...')
while True:
line = yield 'xD' # send value goes there, 'xD' is returned value from generator.
if pattern in line.lower():
print(line)
g = grep('Elo')
next(g) # Coroutines needs priming.
print(g.send('Siema ziomek'))
g.send('Siema elo ziomek')
g.send('xDD')

---

421. What does yield from do?

What yield from does is it establishes a transparent bidirectional connection between the caller and the sub-generator:
- The connection is "transparent" in the sense that it will propagate everything correctly too, not just the elements being generated (e.g. exceptions are propagated).
- The connection is "bidirectional" in the sense that data can be both sent from and to a generator

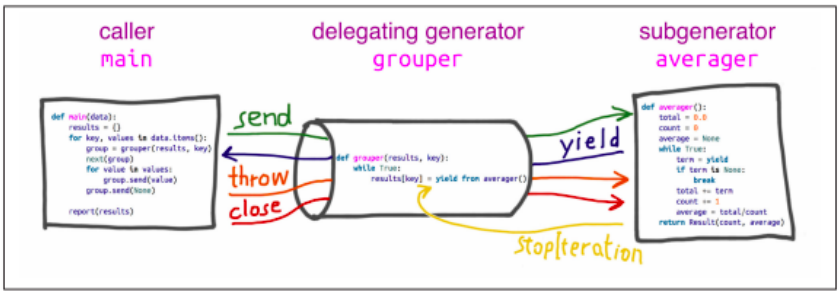

Figure 16-2. While the delegating generator is suspended at `yield from`, the caller sends data directly to the subgenerator, which yields data back to the caller. The delegating generator resumes when the subgenerator returns and the interpreter raises `StopIteration` with the returned value attached[7].

---

422. What is GIL?

Global Interpreter Lock.
The CPython interpreter is not thread-safe internally, so it has a GIL, which allows only one thread at time to execute Python bytecodes. That's why a single Python process usually cannot use multiple CPU cores at the same time.
Simplifies many low-level details (memory management, callouts to C extensions)
(IronPython and jython are not limited that way)

In CPython, the global interpreter lock, or GIL, is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once. This lock is necessary mainly because CPython's memory management is not thread-safe. (However, since the GIL exists, other features have grown to depend on the guarantees that it enforces.)
The GIL is controversial because it prevents multithreaded CPython programs from taking full advantage of multiprocessor systems in certain situations. Note that potentially blocking or long-running operations, such as I/O, image processing, and NumPy number crunching, happen outside the GIL.

---

423. What Pythonic means?

Exploiting the features of the Python language to produce code that is clear, concise and maintainable.
Pythonic means code that doesn't just get the syntax right but that follows the conventions of the Python community and uses the language in the way it is intended to be used.

---

424. What is descriptor?

Descriptiors are a way of reusing the same access logic in multiple attributes. Example: field types in ORMs like Django and SQLAlchemy.
It's a class that implements __get__, __set__ and __delete__ methods.
The property class implements the fully descriptor protocol.
If an object defines both __get__() and __set__(), it is considered a data descriptor
Descriptors that only define __get__() are called non-data descriptors (they are typically used for methods but other uses are possible).

---

425. Old-Style vs. New-Style Classes

With old-style classes, class and type are not quite the same thing. An instance of an old-style class is always implemented from a single built-in type called instance. If obj is an instance of an old-style class, obj.__class__ designates the class, but type(obj) is always instance. The following example is taken from Python 2.7:

```
>>> class Foo:
... pass
...
>>> x = Foo()
>>> x.__class__
<class __main__.Foo at 0x000000000535CC48>
>>> type(x)
<type 'instance'>
```

New-style classes unify the concepts of class and type. If obj is an instance of a new-style class, type(obj) is the same as obj.__class__:

```
>>> class Foo:
... pass
>>> obj = Foo()
>>> obj.__class__
<class '__main__.Foo'>
>>> type(obj)
<class '__main__.Foo'>
>>> obj.__class__ is type(obj)
True
```

---

426. What is type?

type is a metaclass, of which classes are instances. Just as an ordinary object is an instance of a class, any new-style class in Python, and thus any class in Python 3, is an instance of the type metaclass.
You can also call type() with three arguments—type(<name>, <bases>, <dct>):
<name> specifies the class name. This becomes the __name__ attribute of the class.
<bases> specifies a tuple of the base classes from which the class inherits. This becomes the __bases__ attribute of the class.
<dct> specifies a namespace dictionary containing definitions for the class body. This becomes the __dict__ attribute of the class.
Calling type() in this manner creates a new instance of the type metaclass. In other words, it dynamically creates a new class.

---

427. What is metaclass?

A metaclass is the class of a class. A class defines how an instance of the class (i.e. an object) behaves while a metaclass defines how a class behaves. A class is an instance of a metaclass. To create your own metaclass in Python you really just want to subclass type.
Metaclasses are the 'stuff' that creates classes.
You define classes in order to create objects, right?
But we learned that Python classes are objects.
Well, metaclasses are what create these objects. They are the classes' classes, you can picture them this way:
MyClass = MetaClass()
my_object = MyClass()
You've seen that type lets you do something like this:
MyClass = type('MyClass', (), {})
It's because the function type is in fact a metaclass. type is the metaclass Python uses to create all classes behind the scenes.
