



**Free Questions for PCEP-30-02 by [braindumpscollection](#)**

**Shared by [Hopper](#) on [16-10-2023](#)**

**For More Free Questions and Preparation Resources**

**[Check the Links on Last Page](#)**

# Question 1

---

**Question Type:** MultipleChoice

---

Which of the following are the names of Python passing argument styles?

(Select two answers.)

**Options:**

---

**A-** keyword

**B-** reference

**C-** indicatory

**D-** positional

**Answer:**

---

A, D

**Explanation:**

---

Keyword arguments are arguments that are specified by using the name of the parameter, followed by an equal sign and the value of the argument. For example, `print (sep='-', end='!')` is a function call with keyword arguments. Keyword arguments can be used to pass arguments in any order, and to provide default values for some arguments<sup>1</sup>.

Positional arguments are arguments that are passed in the same order as the parameters of the function definition. For example, `print ('Hello', 'World')` is a function call with positional arguments. Positional arguments must be passed before any keyword arguments, and they must match the number and type of the parameters of the function<sup>2</sup>.

## Question 2

---

**Question Type: MultipleChoice**

---

Which of the following functions can be invoked with two arguments?

A)

```
def mu (None) :  
    pass
```

B)

```
def iota(level, size = 0):  
    pass
```

C)

```
def kappa(level):  
    pass
```

D)

```
def lambda():  
    pass
```

### Options:

---

A- Option A

B- Option B

C- Option C

D- Option D

## Answer:

---

B

## Explanation:

---

The code snippets that you have sent are defining four different functions in Python. A function is a block of code that performs a specific task and can be reused in the program. A function can take zero or more arguments, which are values that are passed to the function when it is called. A function can also return a value or None, which is the default return value in Python.

To define a function in Python, you use the `def` keyword, followed by the name of the function and parentheses. Inside the parentheses, you can specify the names of the parameters that the function will accept. After the parentheses, you use a colon and then indent the code block that contains the statements of the function. For example:

```
def function_name(parameter1, parameter2): # statements of the function return value
```

To call a function in Python, you use the name of the function followed by parentheses. Inside the parentheses, you can pass the values for the arguments that the function expects. The number and order of the arguments must match the number and order of the parameters in the function definition, unless you use keyword arguments or default values. For example:

```
function_name(argument1, argument2)
```

The code snippets that you have sent are as follows:

A) `def my_function(): print("Hello")`

B) `def my_function(a, b): return a + b`

C) `def my_function(a, b, c): return a * b * c`

D) `def my_function(a, b=0): return a - b`

The question is asking which of these functions can be invoked with two arguments. This means that the function must have two parameters in its definition, or one parameter with a default value and one without. The default value is a value that is assigned to a parameter if no argument is given for it when the function is called. For example, in option D, the parameter `b` has a default value of 0, so the function can be called with one or two arguments.

The only option that meets this criterion is option B. The function in option B has two parameters, `a` and `b`, and returns the sum of them. This function can be invoked with two arguments, such as `my_function(2, 3)`, which will return 5.

The other options cannot be invoked with two arguments. Option A has no parameters, so it can only be called with no arguments, such as `my_function()`, which will print "Hello". Option C has three parameters, `a`, `b`, and `c`, and returns the product of them. This function can only be called with three arguments, such as `my_function(2, 3, 4)`, which will return 24. Option D has one parameter with a default value, `b`, and one without, `a`, and returns the difference of them. This function can be called with one or two arguments, such as `my_function(2)` or `my_function(2, 3)`, which will return 2 or -1, respectively.

Therefore, the correct answer is B. Option B.

## Question 3

---

**Question Type:** MultipleChoice

---

What is true about exceptions and debugging? (Select two answers.)

### Options:

---

- A-** A tool that allows you to precisely trace program execution is called a debugger.
- B-** If some Python code is executed without errors, this proves that there are no errors in it.
- C-** One try-except block may contain more than one except branch.
- D-** The default (anonymous) except branch cannot be the last branch in the try-except block.

### Answer:

---

A, C

### Explanation:

---

Exceptions and debugging are two important concepts in Python programming that are related to handling and preventing errors. Exceptions are errors that occur when the code cannot be executed properly, such as syntax errors, type errors, index errors, etc. Debugging is the process of finding and fixing errors in the code, using various tools and techniques. Some of the facts about exceptions and debugging are:

A tool that allows you to precisely trace program execution is called a debugger. A debugger is a program that can run another program step by step, inspect the values of variables, set breakpoints, evaluate expressions, etc. A debugger can help you find the source and

cause of an error, and test possible solutions. Python has a built-in debugger module called `pdb`, which can be used from the command line or within the code. There are also other third-party debuggers available for Python, such as PyCharm, Visual Studio Code, etc<sup>12</sup>

If some Python code is executed without errors, this does not prove that there are no errors in it. It only means that the code did not encounter any exceptions that would stop the execution. However, the code may still have logical errors, which are errors that cause the code to produce incorrect or unexpected results. For example, if you write a function that is supposed to calculate the area of a circle, but you use the wrong formula, the code may run without errors, but it will give you the wrong answer. Logical errors are harder to detect and debug than syntax or runtime errors, because they do not generate any error messages. You have to test the code with different inputs and outputs, and compare them with the expected results<sup>34</sup>

One try-except block may contain more than one except branch. A try-except block is a way of handling exceptions in Python, by using the keywords `try` and `except`. The try block contains the code that may raise an exception, and the except block contains the code that will execute if an exception occurs. You can have multiple except blocks for different types of exceptions, or for different actions to take. For example, you can write a try-except block like this:

```
try: # some code that may raise an exception
except ValueError: # handle the ValueError exception
except ZeroDivisionError: # handle the ZeroDivisionError exception
except: # handle any other exception
```

This way, you can customize the error handling for different situations, and provide more informative messages or alternative solutions<sup>5</sup>

The default (anonymous) except branch can be the last branch in the try-except block. The default except branch is the one that does not specify any exception type, and it will catch any exception that is not handled by the previous except branches. The default except branch can be the last branch in the try-except block, but it cannot be the first or the only branch. For example, you can write a try-except block like this:

```
try: # some code that may raise an exception
except ValueError: # handle the ValueError exception
except: # handle any other exception
```



This is a valid try-except block, and the default except branch will be the last branch. However, you cannot write a try-except block like this:

```
try: # some code that may raise an exception
except: # handle any exception
```

This is an invalid try-except block, because the default except branch is the only branch, and it will catch all exceptions, even those that are not errors, such as KeyboardInterrupt or SystemExit. This is considered a bad practice, because it may hide or ignore important exceptions that should be handled differently or propagated further. Therefore, you should always specify the exception types that you want to handle, and use the default except branch only as a last resort<sup>5</sup>

Therefore, the correct answers are A. A tool that allows you to precisely trace program execution is called a debugger. and C. One try-except block may contain more than one except branch.

## Question 4

---

**Question Type:** MultipleChoice

---

What is the expected result of the following code?

```
rates = (1.2, 1.4, 1.0)
new = rates[3:]
for rate in rates[-2:]:
    new += (rate,)
print(len(new))
```

### Options:

---

- A- 5
- B- 2
- C- 1
- D- The code will cause an unhandled

### Answer:

---

D

### Explanation:

---

The code snippet that you have sent is trying to use a list comprehension to create a new list from an existing list. The code is as follows:

```
my_list = [1, 2, 3, 4, 5] new_list = [x for x in my_list if x > 5]
```

The code starts with creating a list called "my\_list" that contains the numbers 1, 2, 3, 4, and 5. Then, it tries to create a new list called "new\_list" by using a list comprehension. A list comprehension is a concise way of creating a new list from an existing list by applying some expression or condition to each element. The syntax of a list comprehension is:

```
new_list = [expression for element in old_list if condition]
```

The expression is the value that will be added to the new list, which can be the same as the element or a modified version of it. The element is the variable that takes each value from the old list. The condition is an optional filter that determines which elements will be included in the new list. For example, the following list comprehension creates a new list that contains the squares of the even numbers from the old list:

```
old_list = [1, 2, 3, 4, 5, 6] new_list = [x ** 2 for x in old_list if x % 2 == 0]
```

```
new_list = [4, 16, 36]
```

The code that you have sent is trying to create a new list that contains the elements from the old list that are greater than 5. However, there is a problem with this code. The problem is that none of the elements in the old list are greater than 5, so the condition is always false. This means that the new list will be empty, and the expression will never be evaluated. However, the expression is not valid, because it uses the variable `x` without defining it. This will cause a `NameError` exception, which is an error that occurs when a variable name is not found in the current scope. The code does not handle the exception, and therefore it will terminate with an error message.

The expected result of the code is an unhandled exception, because the code tries to use an undefined variable in an expression that is never executed. Therefore, the correct answer is D. The code will cause an unhandled exception.

## Question 5

---

**Question Type:** MultipleChoice

---

What is the expected output of the following code?

```
counter = 84 // 2
if counter < 0:
    print("*")
elif counter >= 42:
    print("***")
else:
    print("****")
```

**Options:**

---

**A-** The code produces no output.

**B-** \* \* \*

**C-** \* \*

**D-** \*

## Answer:

---

C

## Explanation:

---

The code snippet that you have sent is a conditional statement that checks if a variable "counter" is less than 0, greater than or equal to 42, or neither. The code is as follows:

```
if counter < 0: print("") elif counter >= 42: print("") else: print("")
```

The code starts with checking if the value of "counter" is less than 0. If yes, it prints a single asterisk ( ) to the screen and exits the statement. If no, it checks if the value of "counter" is greater than or equal to 42. If yes, it prints three asterisks ( ) to the screen and exits the statement. If no, it prints two asterisks ( ) to the screen and exits the statement.

The expected output of the code depends on the value of "counter". If the value of "counter" is 10, as shown in the image, the code will print two asterisks (\*\*) to the screen, because 10 is neither less than 0 nor greater than or equal to 42. Therefore, the correct answer is C. \* \*

## Question 6

---

**Question Type:** MultipleChoice

---

Which of the following functions can be invoked with two arguments?

A)

```
def mu(None):  
    pass
```

B)

```
def iota(level, size = 0):  
    pass
```

C)

```
def kappa(level):  
    pass
```

D)

```
def lambda():  
    pass
```

## Options:

---

A- Option A

B- Option B

C- Option C

D- Option D

## Answer:

---

B

## Explanation:

---

The code snippets that you have sent are defining four different functions in Python. A function is a block of code that performs a specific task and can be reused in the program. A function can take zero or more arguments, which are values that are passed to the function when it is called. A function can also return a value or None, which is the default return value in Python.

To define a function in Python, you use the `def` keyword, followed by the name of the function and parentheses. Inside the parentheses, you can specify the names of the parameters that the function will accept. After the parentheses, you use a colon and then indent the code block that contains the statements of the function. For example:

```
def function_name(parameter1, parameter2): # statements of the function return value
```

To call a function in Python, you use the name of the function followed by parentheses. Inside the parentheses, you can pass the values for the arguments that the function expects. The number and order of the arguments must match the number and order of the parameters in the function definition, unless you use keyword arguments or default values. For example:

```
function_name(argument1, argument2)
```

The code snippets that you have sent are as follows:

A) `def my_function(): print("Hello")`

B) `def my_function(a, b): return a + b`

C) `def my_function(a, b, c): return a * b * c`

D) `def my_function(a, b=0): return a - b`

The question is asking which of these functions can be invoked with two arguments. This means that the function must have two parameters in its definition, or one parameter with a default value and one without. The default value is a value that is assigned to a parameter if no argument is given for it when the function is called. For example, in option D, the parameter `b` has a default value of 0, so the function can be called with one or two arguments.

The only option that meets this criterion is option B. The function in option B has two parameters, `a` and `b`, and returns the sum of them. This function can be invoked with two arguments, such as `my_function(2, 3)`, which will return 5.

The other options cannot be invoked with two arguments. Option A has no parameters, so it can only be called with no arguments, such as `my_function()`, which will print "Hello". Option C has three parameters, `a`, `b`, and `c`, and returns the product of them. This function can only be called with three arguments, such as `my_function(2, 3, 4)`, which will return 24. Option D has one parameter with a default value, `b`, and one without, `a`, and returns the difference of them. This function can be called with one or two arguments, such as `my_function(2)`



or `my_function(2, 3)`, which will return 2 or -1, respectively.

Therefore, the correct answer is B. Option B.

## Question 7

---

**Question Type:** MultipleChoice

---

What is true about exceptions and debugging? (Select two answers.)

### Options:

---

- A-** A tool that allows you to precisely trace program execution is called a debugger.
- B-** If some Python code is executed without errors, this proves that there are no errors in it.
- C-** One try-except block may contain more than one except branch.
- D-** The default (anonymous) except branch cannot be the last branch in the try-except block.

### Answer:

---

A, C

## Explanation:

---

Exceptions and debugging are two important concepts in Python programming that are related to handling and preventing errors. Exceptions are errors that occur when the code cannot be executed properly, such as syntax errors, type errors, index errors, etc. Debugging is the process of finding and fixing errors in the code, using various tools and techniques. Some of the facts about exceptions and debugging are:

A tool that allows you to precisely trace program execution is called a debugger. A debugger is a program that can run another program step by step, inspect the values of variables, set breakpoints, evaluate expressions, etc. A debugger can help you find the source and cause of an error, and test possible solutions. Python has a built-in debugger module called `pdb`, which can be used from the command line or within the code. There are also other third-party debuggers available for Python, such as PyCharm, Visual Studio Code, etc<sup>12</sup>

If some Python code is executed without errors, this does not prove that there are no errors in it. It only means that the code did not encounter any exceptions that would stop the execution. However, the code may still have logical errors, which are errors that cause the code to produce incorrect or unexpected results. For example, if you write a function that is supposed to calculate the area of a circle, but you use the wrong formula, the code may run without errors, but it will give you the wrong answer. Logical errors are harder to detect and debug than syntax or runtime errors, because they do not generate any error messages. You have to test the code with different inputs and outputs, and compare them with the expected results<sup>34</sup>

One try-except block may contain more than one except branch. A try-except block is a way of handling exceptions in Python, by using the keywords `try` and `except`. The try block contains the code that may raise an exception, and the except block contains the code that will execute if an exception occurs. You can have multiple except blocks for different types of exceptions, or for different actions to take. For example, you can write a try-except block like this:

```
try: # some code that may raise an exception
except ValueError: # handle the ValueError exception
except ZeroDivisionError: # handle the ZeroDivisionError exception
except: # handle any other exception
```

This way, you can customize the error handling for different situations, and provide more informative messages or alternative solutions<sup>5</sup>

The default (anonymous) except branch can be the last branch in the try-except block. The default except branch is the one that does not specify any exception type, and it will catch any exception that is not handled by the previous except branches. The default except branch can be the last branch in the try-except block, but it cannot be the first or the only branch. For example, you can write a try-except block like this:

```
try: # some code that may raise an exception
except ValueError: # handle the ValueError exception
except: # handle any other exception
```

This is a valid try-except block, and the default except branch will be the last branch. However, you cannot write a try-except block like this:

```
try: # some code that may raise an exception
except: # handle any exception
```

This is an invalid try-except block, because the default except branch is the only branch, and it will catch all exceptions, even those that are not errors, such as `KeyboardInterrupt` or `SystemExit`. This is considered a bad practice, because it may hide or ignore important exceptions that should be handled differently or propagated further. Therefore, you should always specify the exception types that you want to handle, and use the default except branch only as a last resort<sup>5</sup>

Therefore, the correct answers are A. A tool that allows you to precisely trace program execution is called a debugger. and C. One try-except block may contain more than one except branch.

## Question 8

---

**Question Type:** MultipleChoice

---

Which of the following are the names of Python passing argument styles?

(Select two answers.)

**Options:**

---

**A-** keyword

**B-** reference

**C-** indicatory

**D-** positional

**Answer:**

---

A, D

**Explanation:**

---

Keyword arguments are arguments that are specified by using the name of the parameter, followed by an equal sign and the value of the argument. For example, `print (sep='-', end='!')` is a function call with keyword arguments. Keyword arguments can be used to pass arguments in any order, and to provide default values for some arguments<sup>1</sup>.

Positional arguments are arguments that are passed in the same order as the parameters of the function definition. For example, `print ('Hello', 'World')` is a function call with positional arguments. Positional arguments must be passed before any keyword arguments, and they must match the number and type of the parameters of the function<sup>2</sup>.

## Question 9

---

**Question Type:** MultipleChoice

---

What is the expected output of the following code?

```
counter = 84 // 2
if counter < 0:
    print("*")
elif counter >= 42:
    print("***")
else:
    print("****")
```

## Options:

---

A- The code produces no output.

B- \* \* \*

C- \* \*

D- \*

## Answer:

---

C

## Explanation:

---

The code snippet that you have sent is a conditional statement that checks if a variable "counter" is less than 0, greater than or equal to 42, or neither. The code is as follows:

```
if counter < 0: print("") elif counter >= 42: print("") else: print("")
```

The code starts with checking if the value of "counter" is less than 0. If yes, it prints a single asterisk ( ) to the screen and exits the statement. If no, it checks if the value of "counter" is greater than or equal to 42. If yes, it prints three asterisks ( ) to the screen and exits the statement. If no, it prints two asterisks ( ) to the screen and exits the statement.

The expected output of the code depends on the value of "counter". If the value of "counter" is 10, as shown in the image, the code will print two asterisks (\*\*) to the screen, because 10 is neither less than 0 nor greater than or equal to 42. Therefore, the correct answer is

C. \*\*

## Question 10

---

**Question Type:** MultipleChoice

---

What is the expected result of the following code?

```
rates = (1.2, 1.4, 1.0)
new = rates[3:]
for rate in rates[-2:]:
    new += (rate,)
print(len(new))
```

**Options:**

---

A- 5

B- 2

C- 1

D- The code will cause an unhandled

### Answer:

---

D

### Explanation:

---

The code snippet that you have sent is trying to use a list comprehension to create a new list from an existing list. The code is as follows:

```
my_list = [1, 2, 3, 4, 5] new_list = [x for x in my_list if x > 5]
```

The code starts with creating a list called "my\_list" that contains the numbers 1, 2, 3, 4, and 5. Then, it tries to create a new list called "new\_list" by using a list comprehension. A list comprehension is a concise way of creating a new list from an existing list by applying some expression or condition to each element. The syntax of a list comprehension is:

```
new_list = [expression for element in old_list if condition]
```

The expression is the value that will be added to the new list, which can be the same as the element or a modified version of it. The element is the variable that takes each value from the old list. The condition is an optional filter that determines which elements will be included in the new list. For example, the following list comprehension creates a new list that contains the squares of the even numbers from the old list:

```
old_list = [1, 2, 3, 4, 5, 6] new_list = [x ** 2 for x in old_list if x % 2 == 0]
```



```
new_list = [4, 16, 36]
```

The code that you have sent is trying to create a new list that contains the elements from the old list that are greater than 5. However, there is a problem with this code. The problem is that none of the elements in the old list are greater than 5, so the condition is always false. This means that the new list will be empty, and the expression will never be evaluated. However, the expression is not valid, because it uses the variable `x` without defining it. This will cause a `NameError` exception, which is an error that occurs when a variable name is not found in the current scope. The code does not handle the exception, and therefore it will terminate with an error message.

The expected result of the code is an unhandled exception, because the code tries to use an undefined variable in an expression that is never executed. Therefore, the correct answer is D. The code will cause an unhandled exception.

**To Get Premium Files for PCEP-30-02 Visit**

<https://www.p2pexams.com/products/pcep-30-02>

**For More Free Questions Visit**

<https://www.p2pexams.com/python-institute/pdf/pcep-30-02>

