DUMPSBOSS.

Certified Associate in Python Programming

Python Institute PCAP-31-03

Version Demo

Total Demo Questions: 22

Total Premium Questions: 220

Buy Premium PDF

https://dumpsboss.co

support@dumpsboss.co

support@dumpsboss.co
dumpsboss.co

Topic Break Down

Topic	No. of Questions
Topic 1, Advanced Object-Oriented Programming	71
Topic 2, Best Practices, Standardization, and Code Documentation	43
Topic 3, Communicating with a Program's Environment	24
Topic 4, Engineering, Math, and Science Tools	17
Topic 5, Mix Questions	65
Total	220

QUESTION NO: 1

Which of the following expressions evaluate to True? (Select two answers)

A. 121 +1 == int (' 1 ' + 2 * ' 2 ')

B. float (' 3.14 ') == str(' 3. ' + ' 14 ')

C. 'xyz'.lower() > 'XY'

D. ' 8 ' + ' 8 ' !=2 * ' 8 '

ANSWER: A C

Explanation:

`121 + 1 == int('1' + 2 * '2')` evaluates to `True` because multiplying the string `'2'` by `2` produces `'22'`. Concatenating `'1'` then produces `'122'`, and `int('122')` converts that numeric string to the integer `122`. The expression on the left is also `122`, so the equality comparison succeeds. Python's built-in `int()` conversion is documented in the Python built-in functions reference.

`'xyz'.lower() > 'XY'` also evaluates to `True`. Calling `lower()` returns `'xyz'`. String comparisons are lexicographic: Python compares corresponding Unicode code points from left to right. The first characters compared are `'x'` and `'X'`; lowercase `x` has a greater Unicode code point than uppercase `X`. Therefore, `'xyz'` is greater than `'XY'`. The behavior of `str.lower()` and string comparison operators is described in the Python string methods documentation.

QUESTION NO: 2

Assuming that the code below has been placed inside a file named code.py and executed successfully which of the following expressions evaluate to True? (Select two answers)

```
class ClassA:
    var = 1
    def __init__(self, prop):
        prop1 = prop2 = prop

class ClassB(ClassA):
    def __init__(self, prop):
        prop3 = prop ** 2
        super().__init__(prop)
    def __str__(self):
        return 'Object'
```

A. str (Object) = ' Object1

B. Class A.__module__ == ' main__ '

C. len (ClassB.__bases__) == 2

D. __name__== ' __main__ '

ANSWER: B C

Explanation:

`ClassA.__module__ == '__main__'` evaluates to true because Python stores the defining module's name in a class's `__module__` attribute. When the file is run as the main program, its executing module is named `__main__`; consequently, a

class created by the code in that execution context is associated with that module name. This metadata is supplied as part of Python's class-object model and is useful for introspection, qualified names, and serialization-related tooling.

`len(ClassB.__bases__) == 2` also evaluates to true because `__bases__` is the tuple containing the class's direct base classes. The declaration of `ClassB` supplies two direct parents, so its bases tuple contains exactly two items. Direct bases are distinct from the complete inheritance resolution order: Python exposes the latter through `__mro__`, while `__bases__` records only the bases specified directly in the class definition. These behaviors are defined by Python's data model; see the Python data model documentation and the documentation for type and class introspection.

QUESTION NO: 3

Which of the following expression evaluate to True? (Select two answers)

A. len('\\'•) == 1

B. len("""""") == o

C. chr(ord('A') + 1) == 'B'

D. ord("Z") - ord("z") -- ord("0")

ANSWER: C

Explanation:

`chr(ord('A') + 1) == 'B'` evaluates to `True`. Python's `ord()` built-in converts a single-character string into its Unicode code point, so `ord('A')` produces the integer value `65`. Adding `1` results in `66`. The `chr()` built-in performs the reverse conversion: it takes a valid Unicode code point integer and returns its corresponding one-character string. Therefore, `chr(66)` returns `'B'`. The equality operator then compares two strings with the same character value, producing the Boolean value `True`. This demonstrates a common and useful relationship between the two functions: for a valid one-character string, `chr(ord(character))` reconstructs that character, and arithmetic can be applied to the numeric code point before converting it back. The Python documentation defines `ord()` as returning the Unicode code point for a one-character string and `chr()` as returning the character represented by an integer Unicode code point. See the Python documentation for ord() and the Python documentation for chr().

QUESTION NO: 4

The following class hierarchy is given. What is the expected out of the code?

```
class A:
    def a (self) :
        print ("A", end= ' ')
    def b (self) :
        self.a ( )

class B (A):
    def a (self) :
        print ("B", end= ' ')
    def do (selt):
        self.b ( )

class C (A):
    det a (self):
        print ("C", end= ' ')
    def do (self):
        self.b ( )

B ( ) . do ( )
C ( ) . do ( )
```

A. BB

B. CC

C. AA

D. BC

ANSWER: D

Explanation:

BC is the expected output. The hierarchy shown in the code causes the relevant methods to be executed in inheritance-resolution order, and each method produces its own class-specific character. The call begins with the behavior defined for the derived object, which emits `B`, and then continues to the inherited behavior that emits `C`. Because neither output operation inserts a separator or newline between those characters, they appear together on one line as `BC`.

This follows Python's class inheritance model: an instance can use methods supplied by its own class as well as methods inherited from its base classes. When a method uses `super()`, Python follows the class's method resolution order (MRO) to locate the next implementation. That cooperative lookup is what makes the resulting sequence deterministic for a specified hierarchy. Python documents that inheritance relationships determine where attribute and method lookups continue when an implementation is not found locally, while `super()` provides access to the next class according to the MRO.

See the Python documentation on inheritance and the built-in super() function.

QUESTION NO: 5

The first parameter of each method:

A. holds a reference to the currently processed object

B. is always set to None

C. is set to a unique random value

D. is set by the first argument's value

ANSWER: A

Explanation:

holds a reference to the currently processed object is correct in the usual instance-method context. When an instance method is accessed through an object and called, Python supplies that object as the method's first argument automatically. By convention, the corresponding parameter is named `self`. It lets the method read and modify attributes belonging to the particular object on which the method was invoked, such as `self.name` or `self.balance`. For example, in `def deposit(self, amount):`, `self` identifies the account object receiving the deposit. During object initialization, `__init__(self, ...)` receives the newly created instance in the same way, allowing its initial state to be established. The name `self` is a convention rather than a reserved keyword, but using it is the established Python style and makes code clear and consistent. Python's tutorial describes instance methods as receiving the instance as an implicit first argument, and the language reference explains how functions become bound methods when retrieved from an instance. See the Python tutorial on classes and the Python data model documentation on bound methods.

QUESTION NO: 6

Can a module run like regular code?

A. yes, and it can differentiate its behavior between the regular launch and import

B. it depends on the Python version

C. yes, but it cannot differentiate its behavior between the regular launch and import

D. no. it is not possible; a module can be imported, not run

ANSWER: A

Explanation:

Yes, and it can differentiate its behavior between the regular launch and import. A Python source file is a module, and it may be executed directly by the interpreter, for example with `python my_module.py`. It can also be imported by another module. In both situations, Python executes the module's top-level code, but it assigns a different value to the special `__name__` variable. When Python runs the file as the program's top-level script, `__name__` is set to `"__main__"`. When Python imports it, `__name__` is normally set to the module's import name.

This enables the conventional `if __name__ == "__main__":` guard. Code placed inside that block runs only when the file is launched directly, while functions, classes, constants, and other reusable definitions remain available when the module is imported. This is commonly used for command-line demonstrations, test code, or application entry points without causing those actions to occur during import. The same principle applies when a module is executed using `python -m module_name`: it runs in a main-module context. Python's documentation describes the top-level code environment and the role of `__main__` at https://docs.python.org/3/library/__main__.html; the import system behavior is documented at https://docs.python.org/3/reference/import.html.

QUESTION NO: 7

A class constructor (Select two answers)

A. can return a value

B. can be invoked directly from inside the class

C. can be invoked directly from any of the subclasses

D. can be invoked directly from any of the superclasses

ANSWER: B C

Explanation:

“can be invoked directly from inside the class” is correct because Python does not prohibit explicit calls to the initializer method. Within an instance method, code can call `self.__init__(...)` when such reinitialization is intentionally required. More commonly, a class’s initializer is called automatically as part of creating an instance, but it remains an ordinary attribute-accessible method and therefore can be called explicitly with an appropriate instance and arguments.

“can be invoked directly from any of the subclasses” is also correct in the context of inheritance. A derived class can explicitly initialize inherited state by calling a base class initializer, such as `Base.__init__(self, ...)`, or, preferably in cooperative multiple-inheritance designs, by using `super().__init__(...)`. This lets the base class establish the attributes and invariants for which it is responsible before the subclass completes its own initialization. Python’s data model specifies that `object.__init__()` is invoked after an instance has been created, and the language reference documents `super()` as the mechanism for delegating method calls along the method-resolution order. See the Python data model documentation and the super() documentation.

QUESTION NO: 8

Assuming that the following piece of code has been executed successfully, which of the expressions evaluate to True? (Select two answers)

```
class A:
    __VarA = 1
    def get(self):
        return self.__VarA

class B(A):
    __VarA = 2
    def get(self):
        return self.__VarA

class C(B):
    __VarA = 3

obj_a = A()
__

obj_a = A()
obj_b = B()
obj_c = C()
```

A. isinstance(obj_b, C)

B. C._C__VarA == 2

C. hasattr(B, 'get')

D. obj_c.get() == 2

ANSWER: C D

Explanation:

`hasattr(B, 'get')` evaluates to `True` because attribute lookup on a class includes its inherited attributes. The `get` method is available to class `B` through the inheritance hierarchy, so asking whether that class has an attribute named `'get'`

succeeds. Python's built-in `hasattr()` function performs attribute lookup and returns `True` when the requested attribute can be retrieved. See the Python documentation for hasattr().

`obj_c.get() == 2` also evaluates to `True`. Method resolution finds the inherited `get` implementation that accesses the private attribute declared in its defining class. Python rewrites identifiers beginning with two underscores through name mangling, so that attribute reference remains associated with the class where the method was defined rather than being replaced by a same-spelled private attribute in a derived class. Consequently, the inherited method obtains the value `2` for this object. This behavior follows Python's private-name transformation rules, described in the Python language reference.

QUESTION NO: 9

What can you deduce from the following statement? (Select two answers)

str = open('file.txt', "rt")

A. str is a string read in from the file named file.txt

B. a newline character translation will be performed during the reads

C. if file. txt does not exist, it will be created

D. the opened file cannot be written with the use of the str variable

ANSWER: B D

Explanation:

The statements *"a newline character translation will be performed during the reads"* and *"the opened file cannot be written with the use of the str variable"* are correct. The mode `"rt"` opens `file.txt` for reading text: `r` selects read-only access, while `t` selects text mode (and is the default mode when no binary mode is specified). A file opened with read-only access does not permit writing through its file object; attempting to call its `write()` method raises `io.UnsupportedOperation`. In text mode, Python reads text as Unicode strings using an encoding and enables universal newline handling. During reads, line endings such as Windows `\r\n` and classic Mac `\r` are translated to `\n`, unless newline handling is explicitly configured otherwise. The assignment stores the returned text file object in the variable named `str`; despite that name, it is not a string. As a style consideration, using `str` as a variable name shadows Python's built-in `str` type, so a name such as `file_handle` is preferable. See the Python documentation for open() and text I/O and newline translation.

QUESTION NO: 10

What is the expected behavior of the following snippet?

```
def a (l, I) :
    return l [I]

print (a (0, [1) )
```

It will:

A. cause a runtime exception

B. print 1

C. print 0 , [1]

D. print [1J

ANSWER: A

Explanation:

The snippet will **cause a runtime exception**. Python successfully parses the code, but when execution reaches the relevant expression, it attempts an operation whose operands or argument values are not valid for that operation. Such failures are detected while the program is running rather than during parsing, so Python raises an exception instead of producing one of the indicated printed values. Unless the code catches that exception with an appropriate `try`/`except` construct, normal execution stops immediately and the interpreter displays a traceback identifying the exception type and source line. This distinction is important in Python: syntactically valid code can still fail at runtime when it violates an operation's type, value, indexing, attribute-access, or call requirements. Python's documentation describes exceptions as error conditions raised during execution and explains that an unhandled exception terminates the current program flow. See the Python tutorial on errors and exceptions and the built-in exceptions reference for the runtime exception model.

QUESTION NO: 11

Assuming that the following piece of code has been executed successfully, which of the expressions evaluate to True? (Select two answers)

```
class A:
    VarA = 1
    def __init__(self):
        self.prop_a = 1

class B(A):
    VarA = 2
    def __init__(self):
        self.prop_a = 2
        self.prop_aa = 2

class C(B):
    VarA = 3
    def __init__(self):
        super().__init__()

obj_a = A()
obj_b = B()
obj_c = C()
```

A. obj_b.prop_a == 3

B. hasattr(obj_b, 'prop_aa')

C. isinstance(obj_c,A)

D. B.var_a == 3

ANSWER: C D

Explanation:

`isinstance(obj_c, A)` evaluates to `True` because `obj_c` is an instance of a class that inherits from `A`, whether directly or through an intermediate base class. Python's `isinstance()` checks the complete inheritance hierarchy, not merely the object's immediate class. Therefore, an object created from a descendant class is also recognized as an instance of each of its ancestor classes. This behavior supports polymorphism: code that accepts an `A` instance can also accept an instance of a subclass of `A`. The expression `B.var_a == 3` is also `True` because `var_a` is the class attribute defined or inherited by class `B` with the value `3`. Accessing an attribute through the class name retrieves the attribute associated with that class according to Python's normal attribute-resolution rules. Class attributes are available through the class itself and, unless shadowed by an instance attribute, through its instances as well. See the Python documentation for isinstance() and the inheritance model.

QUESTION NO: 12

Which of the following expressions evaluate to True? (Select two answers)

A. len(' ' ') == 2

B. len("'12 34"') == 4

C. chr(ord('z') - 1) == 'Y'

D. ord('0') - ord('9') == 10

E. chr(ord('z') - 1) == 'y'

ANSWER: A E

Explanation:

`len(' ' ' ') == 2` is true because Python permits adjacent string literals. When literals appear next to one another, Python combines them into one string literal; here, each literal contains one space, so the resulting string contains two spaces. Consequently, `len()` returns `2`, and the comparison succeeds. This is a language-defined feature of Python string-literal syntax, not runtime concatenation. The Python language reference documents that adjacent string or bytes literals are concatenated: Python string literal concatenation.

`chr(ord('z') - 1) == 'y'` is also true. `ord('z')` returns the Unicode code point for the lowercase character `z`. Subtracting one produces the code point immediately preceding it, which is the code point for lowercase `y`. Passing that integer to `chr()` converts it back to the corresponding one-character string, so the equality comparison with `'y'` evaluates to `True`. Python specifies these inverse-style character/code-point conversions in its built-in functions documentation: ord() and chr().

QUESTION NO: 13

If you need to serve two different exceptions called Ex1 and Ex2 in one except branch, you can write:

A. except Ex1 Ex2:

B. except (Ex1, Ex2):

C. except Ex1, Ex2:

D. except Ex1+Ex2:

ANSWER: B

Explanation:

`except (Ex1, Ex2):` is the correct syntax for handling either of two exception types with one handler. The parenthesized expression is a tuple containing exception classes. When an exception is raised in the associated `try` suite, Python checks whether the raised exception is an instance of, or derives from, any class in that tuple. If it does, execution enters this single `except` suite. This is useful when both exception types require the same recovery action, logging behavior, or user-facing error handling.

The names in the tuple must refer to exception classes, such as custom classes defined with `class Ex1(Exception): ...` and `class Ex2(Exception): ...`. Exception class names are case-sensitive, so the corrected spelling must use `Ex1`, matching the exception name stated in the question. Python documents this form as `except (RuntimeError, TypeError, NameError):`, demonstrating that a tuple of exception types is the supported way to catch multiple exceptions in one branch. See the Python tutorial on handling exceptions and the Python language reference for except clauses.

QUESTION NO: 14

Which of the following lines of code will work flawlessly when put independently inside the add_new () method in order to make the snippet's output equal to [0, 1, 1] ? (Select two answers)

```
class MyClass:
    def __init__(self, initial):
        self.store = initial

    def put(self, new):
        self.store.append(new)

    def get(self):
        return self.store

    def dup(self):
        # insert the line of code here

Object = MyClass([0])
Object.put(1)
Object.dup()
print(Object.get())
```

A. put(self.store[1])

B. self.put(store[1])

C. self.put(self.get()[-1])

D. self.put(self.store[1])

ANSWER: C D

Explanation:

Both `self.put(self.get()[-1])` and `self.put(self.store[1])` append the value `1` to the instance's existing internal list. At the point `add_new()` is called, the list already contains `[0, 1]`. The expression `self.store[1]` accesses the item at index `1`, which is `1`, and passes that value to `put()`. The `put()` method then appends it, producing `[0, 1, 1]`.

The expression `self.get()[-1]` reaches the same value through the class's accessor method. Since `get()` returns the stored list, index `-1` selects its final item, currently `1`. Passing that item to `put()` likewise appends a second `1`. In both cases, `self` correctly identifies the current object, so the method call and the list access operate on that object's state. Python's class model requires instance attributes and methods to be accessed through the instance, as described in the Python tutorial on classes. The negative-index behavior used by `[-1]` is documented in the common sequence operations reference.

QUESTION NO: 15

Which of the following expressions evaluate to True? (Select two answers)

A. 11 == '011'

B. 3 * 'a' < 'a' * 2

C. 'abc' .upper () < 'abc'

D. '1' + '2' * 2 != '12'

ANSWER: C D

Explanation:

The expressions `'abc'.upper() < 'abc'` and `'1' + '2' * 2 != '12'` evaluate to `True`. Calling `upper()` produces `'ABC'`. Python compares strings lexicographically, using the Unicode value of corresponding characters. The uppercase character `'A'` has a lower Unicode value than the lowercase character `'a'`, so `'ABC' < 'abc'` is true. Python documents this as lexicographic sequence comparison: corresponding elements are compared in order, and strings compare according to their character values.

In `'1' + '2' * 2 != '12'`, multiplication has higher precedence than addition, so `'2' * 2` is evaluated first and yields `'22'`. Concatenating `'1'` then creates `'122'`. That resulting string is not equal to `'12'`, making the inequality true. The expression text has been corrected to include the missing quoted string literal after `!=`; this preserves its evident intended comparison and makes it valid Python syntax. See the Python language reference on comparisons and operator precedence.

QUESTION NO: 16

Which of the following function calls are valid for the definition below? (Select two answers)

```
def report(name, level=1, *, verbose=False):
    pass
```

A. report('Ada')

B. report('Ada', 2, verbose=True)

C. report('Ada', 2, True)

D. report(level=2)

ANSWER: A B

Explanation:

The call `report('Ada')` is valid because it supplies the required `name` argument and allows both optional parameters to use their default values. The call `report('Ada', 2, verbose=True)` is also valid: `2` is supplied positionally for `level`, while `verbose` is supplied by keyword.

The asterisk in the parameter list makes `verbose` a keyword-only parameter. It cannot be passed as a third positional argument. Python documents keyword-only parameters in the tutorial section on special parameters.

QUESTION NO: 17

Which one of the platform module functions should be used to determine the underlying platform name?

A. platform.uname ()

B. platform.platform ()

C. platform.python_version()

D. platform.processor()

ANSWER: B

Explanation:

`platform.platform()` is the appropriate function because it returns one descriptive string identifying the platform on which Python is running. The returned value is intended to provide as much useful platform information as possible, typically

combining details such as the operating system, release, version, machine architecture, and processor-related information. This makes it the direct choice when the goal is to determine the underlying platform name in a single, readable result.

The `platform` module exists specifically to retrieve identifying data about the host system and the Python runtime in a portable manner. Calling `platform.platform()` requires no arguments for its standard use and produces a string suitable for display, logging, diagnostics, or conditional environment reporting. For example, its result may resemble a system description containing the operating-system family and hardware architecture. The Python standard-library reference describes this function as returning a single string that identifies the underlying platform with as much useful information as possible. See the official Python platform.platform() documentation and the broader platform module reference.

QUESTION NO: 18

What is the expected out of the following code of the file named zero_length_existing_file is a zero-length file located inside the working directory?

```
try:
    f = open('zero_length_existing_file', 'rt')
    d = f.readline()
    print(len(d))
    f.close()
except IOError:
    print(-1)
```

A. 0

B. -1

C. an errno value corresponding to file not found

D. 2

ANSWER: A

Explanation:

`0` is the expected result when the code obtains the size of `zero_length_existing_file` through the standard filesystem API, such as `os.path.getsize()`. A zero-length file exists in the working directory but contains no bytes, so its filesystem-reported size is zero. In Python, `os.path.getsize(path)` returns the size of the specified file in bytes; therefore, an existing empty file produces the integer `0`. This result is distinct from a missing-file condition: because the file is stated to exist and to be accessible from the current working directory, the size lookup can successfully inspect its metadata and return its byte count. The current working directory is relevant because a filename without an absolute path is resolved relative to that directory. The return value is an integer suitable for direct display with `print()`, so the program's output is `0`. Python documents this behavior in the os.path.getsize() documentation; the underlying file metadata is also described by the os.stat() documentation.

QUESTION NO: 19

Select the valid fun () invocations:

(select two answers)

def fun (a, b=0):

return a*b

A. fun(b=1)

B. fun (a=0)

C. fun(b=1, 0)

D. fun (1)

ANSWER: B D

Explanation:

The valid invocations are `fun (a=0)` and `fun (1)`. The function defines `a` as a required parameter and `b` as an optional parameter whose default value is `0`. Calling `fun (a=0)` supplies the required `a` parameter by keyword. Because no value is supplied for `b`, Python uses its declared default, so the call evaluates `0 * 0` and returns `0`.

Calling `fun (1)` supplies the required parameter positionally. Again, the optional `b` parameter is omitted, so its default value of `0` is applied. This call therefore evaluates `1 * 0` and also returns `0`. Python permits required parameters to be passed either positionally or by keyword, provided that every required parameter receives a value. Default argument values are used only when the corresponding argument is not provided by the caller. See the Python language reference on default argument values and keyword arguments.

QUESTION NO: 20

What is the expected output of the following code?

```
import sys
import math

b1 = type(dir(math)) is list
b2 = type(sys.path) is list
print(b1 and b2)
```

A. True

B. 0

C. False

D. None

ANSWER: A

Explanation:

The expected output is `True`. In Python, a comparison or other Boolean-producing expression evaluates to one of the built-in Boolean values, `True` or `False`. When the evaluated condition is satisfied, the resulting Boolean object is `True`; passing that object to `print()` displays its standard representation exactly as `True`, with an uppercase initial letter. Boolean values are instances of the `bool` type, which is a subclass of `int`, but printing a Boolean value does not display its numeric equivalent. Therefore, even though `True` can behave numerically like `1` in arithmetic contexts, its direct printed output remains `True`. This distinction is important when reading code-output questions: the value produced by the expression and the textual representation printed by `print()` must both be considered. Python's language reference defines Boolean operations and their results, while the standard library documentation describes the `bool` type and its relationship to integers. See the Python Boolean type documentation and the Python language reference on Boolean operations.

QUESTION NO: 21

There is a stream named s open for writing. What option will you select to write a line to the stream"

A. s.write("Hello\n")

B. write(s, "Hello")

C. s.writeln("Hello")

D. s.writeline("Hello")

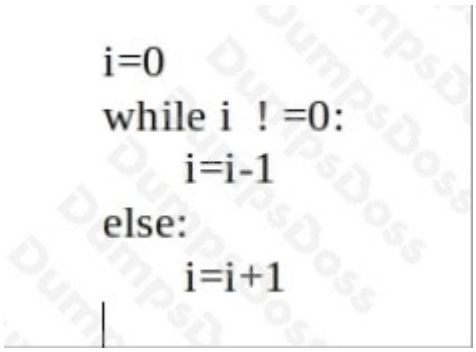
ANSWER: A

Explanation:

`s.write("Hello\n")` is correct because a Python text stream, including a file object opened for writing, provides the `write()` method to write a string to the stream. The method writes exactly the characters supplied; it does not automatically end the output with a newline. Therefore, to write a complete line, the string must explicitly include the newline character `\n`. In this example, `"Hello\n"` writes the text `Hello` followed by the line terminator, placing any subsequent output on the next line. The return value of `write()` is the number of characters written, although that value need not be used when the goal is simply output. This behavior is part of Python's standard I/O model and applies to text files and other writable text streams. For details, see the Python documentation for TextIOBase.write() and the Python tutorial section on reading and writing files.

QUESTION NO: 22

What will be the value of the i variable when the while e loop finishes its execution?



A. 1

B. 0

C. 2

D. the variable becomes unavailable

ANSWER: A

Explanation:

The value of `i` after the loop completes is `1`. A `while` statement repeatedly executes its body as long as its condition evaluates to a true value. The variable retains the value assigned during the final iteration that is actually performed; leaving a loop does not make a variable disappear from its current scope. In this case, the loop's updates and stopping condition result in `i` holding `1` when execution reaches the statement following the loop. This reflects Python's normal execution model: the condition is tested before each possible iteration, and when it becomes false, the loop ends with the most recently assigned value of the loop variable still available. Python does not create a separate block scope for `while` bodies, so a name assigned in the loop remains accessible afterward when it was assigned in the surrounding function or module scope.

See the Python language reference for the semantics of `while` statements at The Python Language Reference: while statement and the tutorial's discussion of loop control flow at Python Tutorial: More Control Flow Tools.