DUMPSBOSS.

PCPP - Certified Professional in Python Programming 1

Python Institute PCPP-32-101

Version Demo

Total Demo Questions: 10

Total Premium Questions: 101

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	33
Topic 2, Best Practices, PEP 8, and Code Conventions	19
Topic 3, GUI Programming	17
Topic 4, Network Programming	7
Topic 5, Working with RESTful APIs	10
Topic 6, Mix Questions	15
Total	101

QUESTION NO: 1

Which of the following is not a widget property?

A. width

B. underline

C. highlightthickness

D. depth

ANSWER: D

Explanation:

`depth` is not a standard Tkinter widget configuration option. Tkinter exposes the configuration system of the underlying Tcl/Tk toolkit: each widget accepts a defined set of option names, which can be supplied at construction time or changed later through methods such as `configure()`. The available names depend somewhat on the widget class, but Tk's documented option database and widget command interfaces define the supported configuration properties. A property named `depth` is not part of that standard option set, so attempting to configure an ordinary Tkinter widget with it would normally result in Tcl/Tk reporting an unknown option. This should not be confused with visual three-dimensional effects in Tk interfaces, which are controlled through supported appearance-related settings such as border width and relief rather than a generic depth property. Python's Tkinter documentation describes how widget options are passed through to Tcl/Tk and inspected or modified with widget configuration methods. The Tcl/Tk documentation provides the authoritative option naming conventions used by Tk widgets. See the Python Tkinter documentation and the Tcl/Tk options reference.

QUESTION NO: 2

Select the true statements about the connection-oriented and connectionless types of communication. (Select two answers.)

A. In the context of TCP/IP networks, the communication side that initiates a connection is called the client, whereas the side that answers the client is called the server.

B. Connectionless communications are usually built on top of TCP.

C. Using walkie-talkies is an example of a connection-oriented communication.

D. A phone call is an example of a connection-oriented communication.

ANSWER: A D

Explanation:

In the context of TCP/IP networks, the communication side that initiates a connection is called the client, whereas the side that answers the client is called the server. This describes the conventional client-server relationship used by many network applications. A client actively requests a service and initiates communication with a server endpoint that listens for and accepts requests. For example, Python's socket documentation shows a client calling `connect()` and a server using `listen()` followed by `accept()` to receive an incoming connection. This arrangement is especially characteristic of connection-oriented stream sockets, commonly implemented with TCP. See the Python socket module documentation.

A phone call is an example of a connection-oriented communication because communication begins only after a call setup process establishes an end-to-end session between the caller and recipient. The participants then exchange voice data during that established session, and the session is explicitly ended when the call finishes. This is a useful real-world analogy for TCP's connection-oriented model: TCP establishes a connection before application data is exchanged and provides a reliable, ordered byte stream for the duration of that connection. The Internet Engineering Task Force's TCP specification, RFC 9293 describes TCP as a connection-oriented transport protocol and defines its connection establishment and termination behavior.

QUESTION NO: 3

Select the true statement about the___name___attribute.

A. which a class instance belongs.

B. ___name is a special attribute, which is inherent for both classes and instances, and it contains a dictionary of object attributes.

C. belongs.

D. __name__ is a special attribute, which is inherent for classes, and it contains the name of a class.

ANSWER: D

Explanation:

`__name__ is a special attribute, which is inherent for classes, and it contains the name of a class.` is correct. When Python executes a `class` statement, it creates a class object and assigns that object's name to its `__name__` attribute as a string. For example, after `class Vehicle: pass`, evaluating `Vehicle.__name__` produces `'Vehicle'`. This is useful in introspection, logging, debugging, registries, and code that needs to identify a class dynamically without hard-coding its name. The value represents the identifier used when the class was defined; it is metadata associated with the class object rather than a collection of the object's ordinary attributes. Python's data model documents `__name__` among the special attributes of class objects, and the language tutorial explains that a class definition creates a class object to which attributes can be attached and accessed with dot notation. The same spelling is also used in other Python contexts, including modules and functions, but in the context of a class it specifically provides that class's name. See the Python data model documentation for type. __name__ and the Python tutorial's class-object documentation.

QUESTION NO: 4

Select the true statements about the sqirte3 module. (Select two answers.)

A. The sqlite3 module provides an interface compliant with the DB-API 2.0.
The sqlite3 module in python provides an interface compliant to the DB-API 2.0. Thus, it follows a standard performance metric that allows for consistency in database programming with python.

B. The special name memory is used to create a database in RAM.
The special name 'memory' is used to create a database in RAM using the sqlite3 module. Thus, when you use it as the name of the database file while opening a connection, it creates a temporary database that exists only in memory.
Reference: Official Python documentation on sqlite3: https://docs.python.org/3/library/sqlite3.html

C. The sqhte3 module does not support transactions.

D. The fetchall method returns an empty list when no rows are available

ANSWER: A D

Explanation:

"The sqlite3 module provides an interface compliant with the DB-API 2.0." is correct because Python's standard-library `sqlite3` module is explicitly documented as a DB-API 2.0 interface for SQLite databases. DB-API 2.0 defines a common programming interface for Python database modules, including connection objects, cursor objects, parameter substitution, exception classes, and transaction-related behavior. This lets code use familiar database interaction patterns while working with SQLite. The standard does not define a performance metric; its purpose is interface consistency and portability. See the Python sqlite3 documentation and PEP 249, Python Database API Specification v2.0.

"The fetchall method returns an empty list when no rows are available" is also correct. A cursor's `fetchall()` method retrieves all remaining rows from the result set and returns them as a list. When the query produced no rows, or when all rows have already been fetched, there are no remaining rows to retrieve, so the returned value is an empty list. This behavior enables callers to iterate over the result consistently without requiring a separate null-value check.

QUESTION NO: 5

Select the true statements about TCP socket communication. (Select two answers.)

A. A server socket normally calls `listen()` before accepting connections.

B. `accept()` returns a new socket for communication with a connected client.

C. TCP preserves the boundaries of every call to `send()`.

D. A TCP client must call `listen()` before calling `connect()`.

ANSWER: A B

Explanation:

TCP is a connection-oriented protocol. A server commonly calls `bind()` to associate its socket with a local address, then calls `listen()` to accept incoming connection requests. Calling `accept()` returns a new socket used for communication with one connected client; the original listening socket remains available to accept further connections. TCP provides a reliable ordered byte stream, but it does not preserve application message boundaries. See the Python socket module documentation and RFC 9293.

QUESTION NO: 6

Select the true statement about the___name___attribute.

A. ___name___is a special attribute, which is inherent for both classes and instances, and it contains information about the class to

B. ___name is a special attribute, which is inherent for both classes and instances, and it contains a dictionary of object attributes.

C. __name___is a special attribute, which is inherent for classes and it contains information about the class to which a class instance

D. __name__ is a special attribute, which is inherent for classes, and it contains the name of a class.

ANSWER: D

Explanation:

The statement "`__name__` is a special attribute, which is inherent for classes, and it contains the name of a class." is correct. In Python, a class object is created with a `__name__` attribute whose value is the class name as a string. For example, after defining `class Account: pass`, evaluating `Account.__name__` produces `'Account'`. This is useful for introspection, diagnostics, logging, debugging, and frameworks that need to identify classes dynamically without hard-coding their names. The attribute belongs to the class object itself, so it can be read directly from the class. An instance can generally access the value through normal attribute lookup as well, but the attribute represents metadata of the class rather than an instance-specific value. Python's data model describes classes as objects with a namespace and identifies `__name__` as the class name; the built-in `type` documentation also demonstrates that class objects expose this attribute. See the Python data model documentation and the documentation for type().

QUESTION NO: 7

Analyze the following snippet and select the statement that best describes it.

```
def f1(*arg, **args):
    pass
```

A. The code is syntactically correct despite the fact that the names of the function parameters do not follow the naming convention

B. The `*arg` parameter holds positional arguments, while the `**kwarg` parameter holds named arguments.

C. The code is missing a placeholder for unnamed parameters.

D. The code is syntactically incorrect - the function should be defined as def f1 (*args, **kwargs) :

ANSWER: A

Explanation:

The code is syntactically correct despite the fact that the names of the function parameters do not follow the naming convention. In a function definition, the identifiers following `*` and `**` are ordinary parameter names selected by the programmer. Thus, a definition such as `def f1(*arg, **kwarg):` is valid Python syntax. The single asterisk makes `arg` a variadic positional parameter: any extra positional arguments are collected into a tuple bound to that name. The double asterisk makes `kwarg` a variadic keyword parameter: extra keyword arguments are collected into a dictionary bound to that name. Although `args` and `kwargs` are widely used conventional names, Python does not require those particular identifiers. The language behavior depends on the asterisk markers, not on whether the parameter names are plural or match the customary spelling. Consequently, using `arg` and `kwarg` does not make the definition invalid. Python's language reference describes how a starred parameter collects excess positional arguments and a double-starred parameter collects excess keyword arguments; see the Python function definitions reference. The naming convention is also reflected in the PEP 8 naming guidance, which provides style recommendations rather than syntax requirements.

QUESTION NO: 8

Which sentence about the ©property decorator is false?

A. The @property decorator should be defined after the method that is responsible for setting an encapsulated attribute.

B. The @property decorator designates a method which is responsible for returning an attribute value

C. The @property decorator marks the method whose name will be used as the name of the instance attribute

D. The @property decorator should be defined before the methods that are responsible for setting and deleting an encapsulated attribute

ANSWER: A

Explanation:

The statement "The @property decorator should be defined after the method that is responsible for setting an encapsulated attribute" is false because a property's getter must first create the `property` object in the class namespace. A setter is then attached to that existing property by decorating a method with the getter property's `.setter` method, conventionally written as `@attribute_name.setter`. Consequently, the getter decorated with `@property` must appear earlier in the class body than the corresponding setter. This ordering is required by how the decorators are evaluated while the class body executes: the setter decorator needs to look up the already-created property using its name. The same pattern applies when adding a deleter through `@attribute_name.deleter`. Together, these methods provide managed attribute access while preserving normal attribute syntax for users of the class. Python's built-in `property` documentation describes the getter, setter, and deleter interfaces, and the language tutorial demonstrates defining the getter before using `@x.setter`. See the Python built-in property documentation and the Python tutorial's properties section.

QUESTION NO: 9

Select the correct statements about the csv module.

(Select two answers.)

A. The DictReader method always gets the header from the first line in the file.

B. A reader object maps each row to a list of strings.

C. It is possible to create a DictWriter object without specifying a header.

D. A DictReader object maps each row to a dict.

ANSWER: B D

Explanation:

A reader object maps each row to a list of strings. The `csv.reader` interface iterates over CSV input and yields a sequence for each record; in ordinary use, each record is represented as a list whose elements are the values from that row. This makes `reader` suitable when columns are accessed by their numeric positions. The CSV module deliberately does not generally infer application-level data types, allowing calling code to perform any necessary conversions after reading values.

A DictReader object maps each row to a dict. `csv.DictReader` uses field names to associate each column value with a key, so iterating over it yields dictionary-based row mappings rather than positional lists. When field names are not supplied to the constructor, they are obtained from the first row of the input; when they are supplied, those names define the mapping. Dictionary rows make code clearer and less fragile when it needs to access values by meaningful column names, such as `row["email"]`. The standard-library documentation describes both reader types and their row representations at Python csv module documentation and DictReader documentation.

QUESTION NO: 10

Select the true statements about the sqirte3 module. (Select two answers.)

A. The sqlite3 module provides an interface compliant with the DB-API 2.0.

B. The special name memory is used to create a database in RAM.

C. The sqhte3 module does not support transactions.

D. The fetchall method returns an empty list when no rows are available

ANSWER: A D

Explanation:

The sqlite3 module provides a DB-API 2.0 interface, which gives Python code a standardized database-programming model. In practical terms, this includes familiar concepts such as connection objects, cursor objects, parameterized SQL execution, transaction control, and result retrieval methods. SQLite is embedded rather than a separate database server, but the Python module still follows the DB-API 2.0 specification where applicable, making its interface consistent with many other Python database drivers.

The fetchall method returns all remaining rows from a query result as a list. Its return type is a list even when the result set has been exhausted or the query produced no rows; in that case, the returned value is an empty list. This behavior is useful because application code can iterate over the result without first needing a special null-value check. The Python documentation explicitly specifies that fetchall returns an empty list when no rows are available. See the Python sqlite3 documentation and PEP 249, Python Database API Specification v2.0.