Test for Python Developer

REDAMP SECURITY s.r.o.

REDAMP

SECURITY

Huynh Thai Hoc

November 16, 2023

Contents

1. Python

Mutable / Immutable: Name at least three mutable and immutable built-in types

No	Mutable	Immutable
1	allows change its values without changing its identity	doesn't allow changes in its value
2	Name: List, Dictionary, Set	Name: Tuple, String, number

What is the difference between list and tuple?

No	List	Tuple
1	Mutable	Immutable
2	Syntax: [ ]	Syntax: ( )
3	List can be modified, such as adding or removing items	Tuples are useful when you want an immutable collection.

What happens if you pass an argument to a function and modify the argument inside? Will the value of this argument be changed after calling this function?

The behavior depends on whether the argument passed to the function is mutable or immutable.

- If you pass a mutable object (such as list, dictionary, set) to a function and modify it inside the function, the changes will affect the original object outside the function. This is because mutable objects can be modified in place.
- If you pass an immutable object (such as number, string, or tuple) to a function and modify it inside the function, the changes will not affect the original value outside the function. This is because immutable objects cannot be modified in place; any operation that appears to modify them creates a new object.
- **Example:**

	Mutable Object	Immutable Object

Code Snippets	def ex_mutable(arr):
	  arr.append(4)
	my_list = [1, 2, 3]
	ex_mutable(my_list)
	print(my_list)
		def ex_immutable(num):
		  num = num + 1
		  return num
		a = 10
		result = ex_immutable(a)
		print("a={},result={}".format(a,result))

Output	[1,2,3,4]	a = 10, result = 11

Shallow / Deep Copy: What is the difference between shallow1 and deep copy2 of an object in Python?

No	Shallow Copy	Deep Copy
1	Stores the references of objects to the original memory address	Stores copies of object's value
2	Changes are reflected in the original object	Doesn't reflect changes in the original object
3	Stores the copy of the original object and points the reference to the objects	Stores the copy of the original object and recursively copies the object
Code Snippets	import copy original_list = [1, [2, 3], 4] shallow_copy = copy.copy(original_list) print(f"Original: {id(original_list)}") print(f"Shallow: {id(shallow_copy)}") print(f"Original: {id(original_list[1])}") print(f"Shallow:{id(shallow_copy[1])}") shallow_copy[1][0] = 'X' print(original_list) print(shallow_copy)	import copy original_list = [1, [2, 3], 4] deep_copy = copy.deepcopy(original_list) print(f"Original: {id(original_list)}") print(f"Shallow: {id(deep_copy)}") print(f"Original: {id(original_list[1])}") print(f"Shallow:{id(deep_copy[1])}") deep_copy[1][0] = 'X' print(original_list) print(deep_copy)
References	Original: 1835784832512 Shallow: 1835784834816 Original: 1835784903552 Shallow: 1835784903552	Original: 2454351368512 Shallow: 2454351349824 Original: 2454351416640 Shallow: 2454351349952
Reflect object	[1, ['X', 3], 4] [1, ['X', 3], 4]	[1, [2, 3], 4] [1, ['X', 3], 4]

Concurrent Computation in Python: What are the possibilities of implementing concurrent computation that Python offers? Briefly describe them.

Concurrency involves allowing multiple tasks to alternate in accessing shared resources, such as disk space, network connections, or a single CPU core. Its goal is to *prevent tasks from blocking* each other.

Python offers several possibilities for implementing concurrent computation:

- **Thread**: A thread is a separate flow of execution. It provides a way to achieve concurrency, allowing multiple operations to appear as if they are making progress simultaneously.

Example:

```
import threading
import time
def task_one(times):
    for _ in range(times):
        print("Task One is running")
        time.sleep(1)
def task_two(times):
    for _ in range(times):
        print("Task Two is running")
        time.sleep(1)
if __name__ =='__main__':
    thread1 = threading.Thread(target=task_one,args=(3,))
    thread2 = threading.Thread(target=task_two,args=(3,))
    thread1.start()
    thread2.start()
    # Wait for both threads to finish
    thread1.join()
    thread2.join()

    print("Both threads have finished")
```

```
Task One is running
Task Two is running
Task Two is running
Task One is running
Task Two is running
Task One is running
Both threads have finished
```

- **Async**: is an alternative approach to running functions concurrently. Instead of relying on system threads, they use special programming constructs. These coroutines are still managed by the Python runtime, but they come with significantly less overhead compared to traditional threads.

  Example:

```
import asyncio
import time
def task_one(times):
    for _ in range(times):
        print("Task One is running")
        time.sleep(1)
def task_two(times):
    for _ in range(times):
        print("Task Two is running")
        time.sleep(1)
# Asynchronous function
async def async_function(name, times, delay):
    print(f"{name} is starting")
    if name.lower() =='task 1':
        task_one(times)
    else:
        task_two(times)
    await asyncio.sleep(delay)
    print(f"{name} is done after {delay} seconds")

# Asynchronous event loop
async def main():
    # Create tasks to run concurrently
    task1 = async_function("Task 1", 3, 2)
    task2 = async_function("Task 2", 3, 1)
    # Run tasks concurrently within the event loop
    await asyncio.gather(task1, task2)
if __name__ == '__main__':
    # Run the asynchronous event loop
    asyncio.run(main())
```

```
Task 1 is starting
Task One is running
Task One is running
Task One is running
Task 2 is starting
Task Two is running
Task Two is running
Task Two is running
Task 1 is done after 2 seconds
Task 2 is done after 1 seconds
```

Design Patterns, Python Idioms, and OOP: Briefly describe singleton design pattern. Provide at least one example of practical usage.

A Singleton pattern in python is a design pattern that allows you to create just one instance of a class, throughout the lifetime of a program. Using a singleton pattern has many benefits. A few of them are:

- To limit concurrent access to a shared resource.
- To create a global point of access for a resource.
- To create just one instance of a class, throughout the lifetime of a program.

Example 1: (Classic Singleton) Classic Singleton creates an instance only if there is no instance created so far; otherwise, it will return the instance that is already created.

```python
class SingletonClass(object):
  def __new__(cls):
    if not hasattr(cls, 'instance'):
      cls.instance = super(SingletonClass, cls).__new__(cls)
    return cls.instance
if __name__ == '__main__':
    singleton = SingletonClass()
    new_singleton = SingletonClass()

    print(singleton is new_singleton)

    singleton.variable = "Singleton Variable"
    print(new_singleton.variable)
```

Output:

True

Singleton Variable

Example 2: (Borg Singleton) Borg singleton is a design pattern in Python that allows state sharing for different instances.

```python
class BorgSingleton:
    _shared_state = {}  # Class variable to store shared state

    def __new__(cls, *args, **kwargs):
        obj = super(BorgSingleton, cls).__new__(cls, *args, **kwargs)
        obj.__dict__ = cls._shared_state  # Share the state among instances
        return obj

# Usage
instance1 = BorgSingleton()
instance1.shared_variable = "Shared Variable from Instance 1"

instance2 = BorgSingleton()
print(instance2.shared_variable)  # Output: Shared Variable from Instance 1

instance2.shared_variable = "Modified by Instance 2"
print(instance1.shared_variable)  # Output: Modified by Instance 2
```

Output:

Shared Variable from Instance 1

Modified by Instance 2

How would you implement an iterator class. What things (e.g. methods) are required?

Implementing an iterator involves creating a class that defines two main methods: __iter__ and __next__.

__iter__ Method:

- This method returns the iterator object itself. It is required for an object to be considered an iterator.
- This method is called when you use the iter() function on an object.

__next__ Method:

- This method returns the next item from the iterator.
- It is called on each iteration using the next() function.

```python
class MyIterator:
    def __init__(self, string_list):
        self.string_list = string_list
        self.current_index = 0
    def __iter__(self):
        return self  # The iterator object is itself
    def __next__(self):
        if self.current_index < len(self.string_list):
            result = self.string_list[self.current_index]
            self.current_index += 1
            return result
        else:
            raise StopIteration  # Signal the end of iteration

if __name__ =='__main__':
    my_string_iterator = MyIterator(["Apple", "Banana", "Cherry"])

    for item in my_string_iterator:
        print(item, end=' ')
```

What is monkey patching?

Monkey patch only refers to dynamic modifications of a class or module at runtime.

```
# Original class
class OriginalClass:
    def original_method(self):
        return "Original behavior"

# Monkey patching - adding a new method
def new_method(self):
    return "Patched behavior"

if __name__ == '__main__':

    OriginalClass.new_method = new_method
    obj = OriginalClass()
    print(obj.original_method())
    print(obj.new_method())
    OriginalClass.original_method = new_method
    print(obj.original_method())
```

Output:

Original behavior

Patched behavior

Patched behavior

What is dependency injection and how would you implement it?

Dependency injection is used to make a class independent of its dependencies or to create a loosely coupled program. Dependency injection is useful for improving the reusability of code.

For example:

- Each car can have a different type of engine, and changes to one car's engine do not affect another car's functionality.

```
class Engine:
    def __init__(self, fuel_type):
        self.fuel_type = fuel_type

    def start(self):
        print(f"The engine is starting. Using {self.fuel_type} fuel.")

class Car:
    def __init__(self, engine):
        self.engine = engine

    def start(self):
        print("Starting the car.")
        self.engine.start()
        print("The car is now running.")
```

- The Car class depends on the Engine class, and the Engine is injected through the constructor.

What is the difference between instance, static, and class method?

	Instance Method	Static Method	Class Method
Definition	method that operates on an instance of a class	does not depend on the instance or class state	method that operates on the class itself
Access	Can access and modify instance attributes and other instance methods	Cannot access or modify instance attributes or class attributes directly	Can access and modify class attributes but not instance attributes directly
Usage	used for operations that involve the instance's state	used for utility functions that don't rely on instance-specific or class-specific data	used for tasks that involve the class as a whole, rather than a specific instance
Decorator	No specific decorator is required	@staticmethod	@classmethod

Example	`class MyClass:`
	`  class_variable = "CV"`
	`  def class_method(cls, new_value):`
	`  cls.class_variable = new_value`
	`obj1 = MyClass()`
	`obj1.class_method(new_value="NewValue")`
	`print(MyClass.class_variable)`
	`print(obj1.class_variable)`
		`class MyClass:`
		`  class_variable = "CV"`
		`  @staticmethod`
		`  def class_method(cls, new_value):`
		`  cls.class_variable = new_value`
		`obj1 = MyClass()`
		`print(MyClass.class_variable)`
		`print(obj1.class_variable)`
		`obj1.class_method(new_value="NewValue")`
			`class MyClass:`
			`  class_variable = "CV"`
			`  @classmethod`
			`  def class_method(cls, new_value):`
			`  cls.class_variable = new_value`
			`obj1 = MyClass()`
			`obj1.class_method(new_value="NewValue")`
			`print(MyClass.class_variable)`
			`print(obj1.class_variable)`

Output	CV NewValue	TypeError obj1.class_method(new_value="NewValue") ")	NewValue NewValue

What are dunder/magic methods? Provide and briefly describe a few of them?

Dunder/magic means double under. This method has two prefixes and suffix underscores in the method name. This is commonly used for operator overloading.

Here are a few dunder/magic methods and brief descriptions of their purposes:

1. **__init__(self, ...):**
   - Called when an object is created. It initializes the object's attributes.
2. **__str__(self), __repr__(self):**
   - __str__: *Defines behavior for when str() is called on an instance of your class*
   - __repr__: ***To** get called by built-int repr() method to return a machine readable representation of a type*.
3. **__getitem__(self, key):**

- Called to implement indexing. Enables accessing elements using square bracket notation.

Example:

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages
    def __str__(self):
        return f"{self.title} by {self.author}"
    def __repr__(self):
        return f"Book('{self.title}', '{self.author}', {self.pages})"
    def __getitem__(self, key):
        if key == 'title':
            return self.title
        elif key == 'author':
            return self.author
        elif key == 'pages':
            return self.pages
        else:
            raise KeyError(f"Invalid key: {key}")
if __name__ =='__main__':
    book = Book(title="Python Programming", author="Guido van Rossum", pages=350)
    print(str(book))
    print(repr(book))
    print(book['title'])
    print(book['author'])
    print(book['pages'])
```

What are, and how would you implement private, protected, and public class attributes in Python? What about class inheritance when using them?

	Public	**Protected**	**Private**
Naming Convention	No prefix	Name with prefix _ (single underscore)	Name with prefix __ (double underscore)
Access Level	Accessible from anywhere	Use within the class and its subclasses. It might still be accessed from outside the class	Use only within the class. Limited access from outside the class
Inheritance	Public attributes are inherited by subclasses	Protected attributes are inherited by subclasses	Private attributes are not directly inherited by subclasses

Public Members:

```
class Student:
    schoolName = 'School'
    def __init__(self, name, age):
        self.name=name
        self.age=age
if __name__ == '__main__':
    std = Student("Steve", 25)
    print(std.schoolName)
    print(std.name, std.age)
    std.age = 20
    print(std.age)
```

Protected Members:

```
class Student:
    _schoolName = 'School'
    def __init__(self,name, age):
        self._name = name
        self._age = age
    @property
    def schoolName(self):
        return self._schoolName
    @schoolName.setter
    def schoolName(self,newSchoolName):
        self._schoolName = newSchoolName
    @property
    def name(self):
        return self._name
    @name.setter
    def name(self,newname):
        self._name = newname
    @property
    def age(self):
        return self._age
    @age.setter
    def age(self,newage):
        self._age = newage
if __name__ == '__main__':
    std = Student("Steve", 25)
    print(std.schoolName)
    print(std.name, std.age)
    std.age = 20
    print(std.age)
```

Private Members:

```
class Student:
    __schoolName = 'School'

    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    @property
    def schoolName(self):
        return self.__schoolName

    @schoolName.setter
    def schoolName(self, newSchoolName):
        self.__schoolName = newSchoolName

    @property
    def name(self):
        return self.__name

    @name.setter
    def name(self, new_name):
        self.__name = new_name

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, new_age):
        self.__age = new_age

if __name__ == '__main__':
    std = Student("Steve", 25)
    print(std.schoolName)
    print(std.name, std.age)
    std.age = 20
    print(std.age)
```

What are context managers and when would you use it?

Context managers are used to set up and tear down temporary contexts, establish and resolve custom settings, and acquire and release resources.

Example:

```
# Using a context manager for file handling
with open("example.txt", "r") as file:
    content = file.read()
    # Code inside the 'with' block has access to the file object

# The file is automatically closed
```

When would you implement a custom exception?

When you want to distinguish between different types of errors in your application and handle them in a specific way.

Example:

```
class InvalidInputError(Exception):
    pass

def process_input(value):
    if something_is_invalid:
        raise InvalidInputError("Something is invalid.")
    # Rest of the processing logic
```

What are decorators? Please provide at least one example of their usage. Is it possible to stack multiple different decorators?

Decorators allow us to wrap another function in order to extend the behaviour of the wrapped function, without permanently modifying it. It is possible to stack multiple different decorators.

Example:

```
def decorator1(func):
    def wrapper():
        print("Decorator 1: Something is happening before the function is called.")
        func()
        print("Decorator 1: Something is happening after the function is called.")
    return wrapper

def decorator2(func):
    def wrapper():
        print("Decorator 2: Something is happening before the function is called.")
        func()
        print("Decorator 2: Something is happening after the function is called.")
    return wrapper

def decorator3(func):
    def wrapper():
        print("Decorator 3: Something is happening before the function is called.")
        func()
        print("Decorator 3: Something is happening after the function is called.")
    return wrapper

@decorator1
@decorator2
@decorator3
def say_hello():
    print("Hello!")

# Calling the decorated function
say_hello()
```

Output:

Decorator 1: Something is happening before the function is called.

Decorator 2: Something is happening before the function is called.

Decorator 3: Something is happening before the function is called.

Hello!

Decorator 3: Something is happening after the function is called.

Decorator 2: Something is happening after the function is called.

Decorator 1: Something is happening after the function is called.

Code Snippets: What should be the output of the following code snippets?

```
print([1, 2, 3][10:])
print(r" ict a\nd media ")
```

Display the results in the following order:

[]

ict a\nd media

```
def extendlist(val, list=[]):
    list.append(val)
    return list


list1 = extendlist(42)
list2 = extendlist(42, [])
list3 = extendlist("item")
print(list1)
print(list2)
print(list3)
```

Display the results in the following order:

[42, 'item']

[42]

[42, 'item']

To avoid the mutable default argument issue, extendlist function should be:

```
def extendlist(val, list=None):
  if list is None:
    list = []
  list.append(val)
  return list
```

Implement a function, that will find all odd integer numbers from interval <1; 100 000>, and stores them into a list.

```
def extract_odd_numbers(start=1, end=100000):
    odd_numbers = list(range(start, end + 1, 2))
    return odd_numbers
```

Implement a function, that generates infinite sequence of odd numbers

```
from itertools import count
def infinite_sequence_odd_number():
    for number in count(1, 2):
        yield number
```

Write a regular expression, that matches protocol, IPv4 address, and port from the string below. There can be any protocol, IPv4 address, and any port on the input. Protocol and port are optional parts and can be missing. For the string below, it must match groups "protocol=udp", "ipv4=127.0.0.1", "port=53"

"udp://127.0.0.1:53"

pattern = re.compile(r'((?P<protocol>[a-zA-Z]+)://)?(?P<ipv4>\d+\.\d+\.\d+\.\d+)(:(?P<port>\d+))?')

```
input_string = "udp://127.0.0.1:53"
pattern = re.compile(r'((?P<protocol>[a-zA-Z]+)://)?(?P<ipv4>\d+\.\d+\.\d+\.\d+)(:(?P<port>\d+))?')
match = pattern.match(input_string)

if match:
    print(f"Protocol: {match.group('protocol')}")
    print(f"IPv4: {match.group('ipv4')}")
    print(f"Port: {match.group('port')}")
else:
    print("No match found.")
```

2. Testing and Code Quality

What is the difference between unit and integration testing?

Unit Testing It is a type of software testing in which a small piece of code is tested to see if the code works as expected. Integration testing, as its name implies, verifies that the interface between two software units or modules works correctly.

No	Unit Testing	Integration Testing
1	Each module of the software is tested separately.	All modules of the software are tested combined
2	tester knows the internal design of the software	tester doesn't know the internal design of the software
3	It is performed first of all testing processes.	It is performed after unit testing and before system testing.
4	Unit testing is white box testing.	Integration testing is black box testing.

5	Unit testing is performed by the developer.	Integration testing is performed by the tester.
6	It tests parts of the project without waiting for others to be completed.	It tests only after the completion of all parts.

What is mocking and what are its benefits during testing?

Mocking is a technique used in testing to replace parts of software with simulated or controlled components. The purpose of mocking is to isolate the code being tested from the actual implementation of certain dependencies. Mock objects or functions are used to mimic the behavior of real objects or functions, allowing developers to control their responses and interactions during testing.

Example:

```
def test_extract_urls_ips(self, mock_requests_get):
    # Mock the response from requests.get
    mock_response = Mock()
    mock_response.status_code = 200
    mock_response.text = "http://103.29.2.134/a-r.m-5.Sakura,123.29.2.124"
    mock_requests_get.return_value = mock_response
    ....
```

This setup allows you to control the behavior of the requests.get function during the test, ensuring that it returns a predefined response (mock_response) rather than making a real HTTP request.

What is white box and black box testing?

- White box testing is a testing technique where the tester has knowledge of the internal workings, structure, and code implementation of the software being tested.
- Black box testing is a testing technique where the tester has no knowledge of the internal workings, implementation details, or code structure of the software being tested

What is static analysis of code (hint: pep8)?

Static analysis of code refers to the process of analyzing the source code of a program without actually executing it. PEP8 covers various aspects of code style, including indentation, naming conventions, spacing, comments, and other stylistic choices. The goal is to make the code more readable and maintainable.

What stands for CI and CD?

CI: stands for Continuous Integration

CD: stands for Continuous Development/Delivery.

3. Databases

What is the difference between *SQL* and *NoSQL* database systems? Name a few candidates from each group:

No	SQL	NoSQL
1	Use a table-based structure	Use a document-oriented structure

2	Follow a predefined schema	Schema-less or have a dynamic schema
3	Is organized in tables, relationships between tables are established by keys	Is organized key-value pairs
4	you can increase the power of a single server (CPU, RAM, SSD) to handle increased load.	you can add more servers to your NoSQL database to handle large amounts of traffic
5	SQL Server, PostgreSQL	MongoDB

What is a database index? What types of database indexes are familiar to you? (Ideally in PostgreSQL)

A database index is a data structure like the index in a book. Its goal is to speed up the data retrieval process by providing a quick reference to where specific information might be found.

I am familiar with B-Tree indexes in PostgreSQL.

What would you do, if you needed to optimize an SQL query?

1. Ensure that the columns used in the WHERE clause, JOIN conditions, and ORDER BY clause are indexed appropriately. This can significantly speed up data retrieval.
2. Filter data as early as possible in the query execution process by using the WHERE clause wisely.
3. If the query is returning a large result set but you only need a portion of it, consider using the LIMIT clause to restrict the number of rows returned.
4. Only select the columns you actually need instead of using SELECT *. This reduces the amount of data that needs to be processed and transmitted, improving performance.
5. Ensure that JOIN operations are necessary and optimize them by using appropriate JOIN types (INNER, LEFT, etc.). Avoid unnecessary JOINs and consider denormalization if appropriate for your use case.
6. If you only need to check for the existence of rows, consider using EXISTS instead of DISTINCT. Similarly, use the IN operator when appropriate.
7. Subqueries can be performance-intensive. Consider rewriting the query using JOINs or other techniques to achieve the same result.
8. Limit the use of GROUP BY and ORDER BY clauses to only the necessary columns. Consider indexing columns used in these clauses.

What is ELT and ETL (hint: extract, load, transform)? What are differences between them?

ETL (Extract, Transform, Load)

1. **Extract:** In ETL, data is first extracted from source systems. Source systems can include various databases, applications, flat files, and other data repositories. The goal is to gather the necessary data from these sources for analysis or reporting.
2. **Transform:** Extracted data is then transformed to meet the requirements of the target system or data warehouse. Transformation involves cleaning, aggregating, filtering, and converting the data into a format that is suitable for analysis.
3. **Load:** The transformed data is loaded into the target system. Loading involves storing the prepared data in a structured manner so that it can be efficiently queried and analyzed.

ELT (Extract, Load, Transform):

1. **Extract:** Similar to ETL, data is initially extracted from source systems. This extraction phase involves pulling data from various sources, including databases, applications, and files.
2. **Load:** After extraction, the data is loaded directly into the target system without significant transformation. The raw data is stored in its original form within the target system, which is typically a data lake or a big data storage environment.
3. **Transform:** Transformation occurs within the target system. Data is transformed on an as-needed basis, often just before analysis or reporting.

The differences between ELT and ETL:

No	ELT	ETL
Sequence	Extract, Transform, Load	Extract, Load, Transform
Transformation Location	Before loading into the target system	Within the target system
Target System	Traditional data warehouse	Big data storage, data lake, cloud storage
Data Storage	Structured data for analysis	Raw data loaded without extensive transformation
Processing Paradigm	Often batch processing	Supports both batch and real-time processing

Both ETL and ELT have their advantages and are suitable for different use cases. ETL is often associated with structured data and traditional data warehousing, while ELT is favored in big data environments where the raw data is stored in its native format, allowing for more flexible and agile data processing.