20 Intermediate Python Questions and Answers

1. What's the difference between `==` and `is` operators?

Answer: `==` compares values for equality, while `is` compares object identity (memory location).

```python
a = [1, 2, 3]
b = [1, 2, 3]
c = a
print(a == b)  # True (same values)
print(a is b)  # False (different objects)
print(a is c)  # True (same object)
```

2. Explain list comprehensions vs generator expressions.

Answer: List comprehensions create a list in memory immediately, while generator expressions create an iterator that yields values on demand.

```python
list_comp = [x**2 for x in range(5)]  # Creates [0, 1, 4, 9, 16]
gen_exp = (x**2 for x in range(5))    # Creates generator object
print(type(list_comp))  # <class 'list'>
print(type(gen_exp))    # <class 'generator'>
```

3. What are decorators and how do they work?

Answer: Decorators are functions that modify or enhance other functions without changing their code.

```python
```

```python
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Before function call")
        result = func(*args, **kwargs)
        print("After function call")
        return result
    return wrapper

@my_decorator
def greet(name):
    return f"Hello, {name}"

greet("Alice")  # Prints before and after messages
```

4. Explain the difference between `*args` and `**kwargs`.

Answer: `*args` collects positional arguments into a tuple, `**kwargs` collects keyword arguments into a dictionary.

```python
def example_func(*args, **kwargs):
    print("args:", args)
    print("kwargs:", kwargs)

example_func(1, 2, 3, name="Alice", age=30)
# args: (1, 2, 3)
# kwargs: {'name': 'Alice', 'age': 30}
```

5. What's the difference between shallow and deep copy?

Answer: Shallow copy creates a new object but references to nested objects remain the same. Deep copy creates completely independent copies.

```python
import copy
original = [[1, 2], [3, 4]]
shallow = copy.copy(original)
deep = copy.deepcopy(original)

original[0][0] = 'X'
print(shallow)  # [['X', 2], [3, 4]] - affected
print(deep)     # [[1, 2], [3, 4]] - not affected
```

6. How do you handle exceptions in Python?

Answer: Use try-except blocks with optional else and finally clauses.

```python
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"Error: {e}")
except Exception as e:
    print(f"Unexpected error: {e}")
else:
    print("No exceptions occurred")
finally:
    print("This always executes")
```

7. What are context managers and how do you create custom ones?

Answer: Context managers handle resource management automatically using `with` statements by implementing `__enter__` and `__exit__` methods.

```python
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.file.close()

# Usage
with FileManager('test.txt', 'w') as f:
    f.write("Hello World")
```

8. What's the difference between `staticmethod`, `classmethod`, and instance methods?

Answer: Instance methods operate on instances, class methods operate on the class itself, static methods are independent utility functions.

```python
class MyClass:
    class_var = "class variable"

    def instance_method(self):
        return f"Instance method called on {self}"

    @classmethod
    def class_method(cls):
        return f"Class method called on {cls.class_var}"

    @staticmethod
    def static_method():
        return "Static method called"
```

9. How do closures work in Python?

Answer: A closure is a function that captures variables from its enclosing scope, even after the outer function returns.

```python
def outer_function(x):
    def inner_function(y):
        return x + y  # x is captured from outer scope
    return inner_function

closure = outer_function(10)
print(closure(5))  # Output: 15
```

10. What are Python's magic methods (dunder methods)?

Answer: Special methods that define how objects behave with built-in operations, surrounded by double underscores.

```python
```

```python
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __str__(self):
        return f"Point({self.x}, {self.y})"

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __len__(self):
        return int((self.x**2 + self.y**2)**0.5)
```

11. Explain generators and the `yield` keyword.

Answer: Generators are functions that return an iterator, yielding values one at a time instead of returning all at once.

```python
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

# Usage
for num in fibonacci(5):
    print(num)  # 0, 1, 1, 2, 3
```

12. What's the difference between `append()` and `extend()` for lists?

Answer: `append()` adds a single element, `extend()` adds all elements from an iterable.

```python
list1 = [1, 2, 3]
list2 = [4, 5]

list1.append(list2)
print(list1)  # [1, 2, 3, [4, 5]]

list1 = [1, 2, 3]
list1.extend(list2)
print(list1)  # [1, 2, 3, 4, 5]
```

13. What's the difference between `__str__` and `__repr__`?

Answer: `__str__` provides human-readable representation, `__repr__` provides unambiguous representation for developers.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name}, {self.age} years old"

    def __repr__(self):
        return f"Person('{self.name}', {self.age})"

p = Person("Alice", 30)
print(str(p))   # Alice, 30 years old
print(repr(p))  # Person('Alice', 30)
```

14. How do you create a singleton class in Python?

Answer: Use `__new__` method to control instance creation.

```python
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

# Usage
s1 = Singleton()
s2 = Singleton()
print(s1 is s2)  # True
```

15. What are lambda functions and when should you use them?

Answer: Lambda functions are anonymous functions for simple operations, best used with `map()`, `filter()`, `sort()`.

```python
# Regular function
def square(x):
    return x**2

# Lambda equivalent
square_lambda = lambda x: x**2

# Common usage
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
even = list(filter(lambda x: x % 2 == 0, numbers))
```

16. What's the difference between `sort()` and `sorted()`?

Answer: `sort()` modifies the original list in-place, `sorted()` returns a new sorted list.

```python
list1 = [3, 1, 4, 1, 5]
list2 = [3, 1, 4, 1, 5]

list1.sort()  # Modifies original
print(list1)  # [1, 1, 3, 4, 5]

sorted_list2 = sorted(list2)  # Returns new list
print(list2)  # [3, 1, 4, 1, 5] (unchanged)
print(sorted_list2)  # [1, 1, 3, 4, 5]
```

17. How do you implement property decorators?

Answer: Use `@property` decorator to create getter/setter methods that work like attributes.

```python
```

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value < 0:
            raise ValueError("Radius cannot be negative")
        self._radius = value

    @property
    def area(self):
        return 3.14159 * self._radius ** 2

c = Circle(5)
print(c.area)  # 78.53975
c.radius = 3   # Uses setter
```

18. What's the Global Interpreter Lock (GIL) and its impact?

Answer: The GIL is a mutex that prevents multiple threads from executing Python bytecode simultaneously. It limits CPU-bound multithreading but doesn't affect I/O-bound operations.

```python
import threading
import time

def cpu_bound_task():
    # This won't benefit from multithreading due to GIL
    total = sum(i*i for i in range(1000000))
    return total

def io_bound_task():
    # This can benefit from multithreading
    time.sleep(1)
    return "completed"
```

19. How do you use `enumerate()` and `zip()` functions?

Answer: `enumerate()` adds counter to iterable, `zip()` combines multiple iterables.

```python
# enumerate()
fruits = ['apple', 'banana', 'orange']
for index, fruit in enumerate(fruits):
    print(f"{index}: {fruit}")

# zip()
names = ['Alice', 'Bob', 'Charlie']
ages = [25, 30, 35]
for name, age in zip(names, ages):
    print(f"{name} is {age} years old")

# zip with unequal lengths
list1 = [1, 2, 3]
list2 = ['a', 'b']
print(list(zip(list1, list2)))  # [(1, 'a'), (2, 'b')]
```

20. What are metaclasses in Python?

Answer: Metaclasses are classes whose instances are classes themselves. They define how classes are created.

```python
```

```python
class SingletonMeta(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]

class Database(metaclass=SingletonMeta):
    def __init__(self):
        self.connection = "Database connection"

# Usage
db1 = Database()
db2 = Database()
print(db1 is db2)  # True

# Simpler metaclass example
class AutoRepr(type):
    def __new__(mcs, name, bases, namespace):
        def __repr__(self):
            attrs = ', '.join(f"{k}={v}" for k, v in self.__dict__.items())
            return f"{name}({attrs})"
        namespace['__repr__'] = __repr__
        return super().__new__(mcs, name, bases, namespace)

class Person(metaclass=AutoRepr):
    def __init__(self, name, age):
        self.name = name
        self.age = age

p = Person("Alice", 30)
print(p)  # Person(name=Alice, age=30)
```