#_ Mastering Python: Important Notes [Part #1]

1. Swap Values

Swap the values of two variables without using a temporary variable:

```
a, b = b, a
```

2. List Comprehensions

Use list comprehensions for concise and readable code:

```
squared_numbers = [x**2 for x in range(1, 11)]
```

3. Multiple Assignments

Assign multiple variables in a single line:

```
x, y, z = 1, 2, 3
```

4. Unpacking

Unpack elements of a list or tuple into separate variables:

```
numbers = [1, 2, 3]
a, b, c = numbers
```

5. Merge Dictionaries

Merge two dictionaries in Python 3.5+:

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
merged_dict = {**dict1, **dict2}
```

6. Ternary Operator

Use a ternary operator for concise conditional expressions:

```
x = 10
result = "positive" if x > 0 else "non-positive"
```

By: Waleed Mousa

7. Default Dictionary

Create a dictionary with default values for missing keys:

```
from collections import defaultdict
d = defaultdict(int)  # Default value for missing keys is 0
```

8. Zip Function

Combine two or more lists using the `zip` function:

```
names = ["Alice", "Bob", "Charlie"]
scores = [85, 92, 78]
students = zip(names, scores)  # [('Alice', 85), ('Bob', 92),
('Charlie', 78)]
```

9. Enumerate Function

Use `enumerate` to get both the index and value of elements in a list:

```
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits):
    print(f"Index {index}: {fruit}")
```

10. `any` and `all`

Check if any or all elements in a sequence satisfy a condition:

```
numbers = [2, 4, 6, 7, 8]
any_even = any(x % 2 == 0 for x in numbers)
all_even = all(x % 2 == 0 for x in numbers)
```

11. One-Liner Function

Create one-liner functions using lambda:

```
square = lambda x: x ** 2
```

By: Waleed Mousa

12. `filter` and `map`

Use `filter` and `map` for compact and expressive list operations:

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

13. `sorted` Function

Sort a list while keeping the original intact:

```
numbers = [3, 1, 4, 1, 5, 9, 2]
sorted_numbers = sorted(numbers)
```

14. Chain Comparison

Chain comparison for cleaner code:

```
x = 10
if 0 < x < 20:
    print("x is in the range (0, 20)")
```

15. Dictionary Comprehensions

Use dictionary comprehensions for concise dictionary creation:

```
squared_dict = {x: x**2 for x in range(1, 11)}
```

16. `zip(*reversed())` to Transpose a Matrix

Transpose a matrix using `zip` and `reversed` functions:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
transposed_matrix = [list(row) for row in zip(*reversed(matrix))]
# Result: [[7, 4, 1], [8, 5, 2], [9, 6, 3]]
```

By: Waleed Mousa

17. `set` for Unique Elements

Use `set` to get unique elements from a list:

```
numbers = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
unique_numbers = list(set(numbers))
```

18. `collections.Counter`

Use `Counter` to count occurrences of elements in a list:

```
from collections import Counter
numbers = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
number_counts = Counter(numbers)  # Output: Counter({4: 4, 3: 3, 2: 2, 1: 1})
```

19. `sum` with a Starting Value

Use `sum` with a starting value for accumulating values:

```
numbers = [1, 2, 3, 4, 5]
sum_with_start = sum(numbers, 10)  # Result: 25 (10 + 1 + 2 + 3 + 4 + 5)
```

20. `collections.defaultdict` with List

Use `defaultdict` with a list as the default factory:

```
from collections import defaultdict

colors = [("apple", "red"), ("banana", "yellow"), ("cherry", "red"), ("orange", "orange")]
color_dict = defaultdict(list)

for fruit, color in colors:
    color_dict[color].append(fruit)

# Output: {'red': ['apple', 'cherry'], 'yellow': ['banana'], 'orange': ['orange']}
```

By: Waleed Mousa

21. The `with` Statement for File Handling

Use the `with` statement for cleaner file handling:

```
with open("file.txt", "r") as file:
    content = file.read()
# No need to manually close the file, it's automatically handled by the
'with' block.
```

22. `isinstance` for Type Checking

Use `isinstance` for type checking instead of using `type`:

```
if isinstance(value, int):
    # Do something with an integer value
```

23. Multiple Inheritance

Use multiple inheritance to create flexible class hierarchies:

```
class A:
    def method(self):
        print("Method from class A")

class B:
    def method(self):
        print("Method from class B")

class C(A, B):
    pass

instance = C()
instance.method()  # Output: "Method from class A"
```

24. Decorators

Decorators allow you to modify or enhance the behavior of functions or methods:

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Something is happening before the function is called.")
        result = func(*args, **kwargs)
        print("Something is happening after the function is called.")
        return result
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

25. functools.partial

Use `functools.partial` to create functions with fixed arguments:

```
import functools

def power(base, exponent):
    return base ** exponent

square = functools.partial(power, exponent=2)
cube = functools.partial(power, exponent=3)

print(square(5))  # Output: 25
print(cube(5))    # Output: 125
```

26. collections.namedtuple

Create simple classes with named fields using `namedtuple`:

```
from collections import namedtuple

Person = namedtuple("Person", ["name", "age"])
person = Person("Alice", 30)
```

By: Waleed Mousa

```
print(person.name)  # Output: "Alice"
print(person.age)   # Output: 30
```

27. globals() and locals()

Access global and local variables dynamically:

```
x = 10

def my_function():
    y = 20
    print(locals())  # Output: {'y': 20}
    print(globals()['x'])  # Output: 10

my_function()
```

28. List Concatenation

Use + to concatenate lists:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
concatenated_list = list1 + list2  # Output: [1, 2, 3, 4, 5, 6]
```

29. String Formatting

Use f-strings (Python 3.6+) or str.format for string formatting:

```
name = "Alice"
age = 30
print(f"My name is {name} and I am {age} years old.")
# Output: "My name is Alice and I am 30 years old."

print("My name is {} and I am {} years old.".format(name, age))
# Output: "My name is Alice and I am 30 years old."
```

30. try, except, and else

Use try, except, and else for more informative error handling:

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"Error: {e}")
else:
    print(f"Result: {result}")
```

31. collections.deque

Use deque for efficient and thread-safe appends and pops from both ends:

```
from collections import deque
queue = deque()
queue.append(1)
queue.append(2)
queue.append(3)
queue.popleft()  # Output: 1
```

32. Shuffling a List

Use random.shuffle to shuffle elements in a list:

```
import random
numbers = [1, 2, 3, 4, 5]
random.shuffle(numbers)
print(numbers)
```

33. Dictionary Get with Default Value

Use get to access dictionary values with a default value if the key is not present:

```
my_dict = {'a': 1, 'b': 2}
value = my_dict.get('c', 0)  # Output: 0
```

34. `dir()` for Object Attributes

Use `dir()` to get a list of attributes and methods of an object:

```
my_list = [1, 2, 3]
attributes = dir(my_list)
print(attributes)
```

35. `sys.argv` for Command-Line Arguments

Access command-line arguments using `sys.argv`:

```
import sys
if len(sys.argv) > 1:
    file_name = sys.argv[1]
    print(f"File name: {file_name}")
```

36. `zip` for Parallel Iteration

Use `zip` for parallel iteration over multiple lists:

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

37. Check for Membership

Check if an element exists in a list or dictionary:

```
numbers = [1, 2, 3, 4, 5]
if 3 in numbers:
    print("3 is in the list.")

my_dict = {'a': 1, 'b': 2}
if 'b' in my_dict:
    print("Key 'b' exists in the dictionary.")
```

38. The `pass` Statement

Use the `pass` statement as a placeholder for code that will be implemented later:

```
def my_function():
    # TODO: Implement this function
    pass
```

39. `reversed` Function

Reverse a list or string using `reversed`:

```
numbers = [1, 2, 3, 4, 5]
reversed_numbers = list(reversed(numbers))
```

40. FizzBuzz

The classic FizzBuzz problem using a list comprehension:

```
result = ["Fizz" * (x % 3 == 0) + "Buzz" * (x % 5 == 0) or x for x in range(1, 101)]
```

41. Filter False Values

Filter out False values from a list using `filter`:

```
data = [0, 1, False, True, "", "hello", None]
filtered_data = list(filter(None, data))  # Output: [1, True, 'hello']
```

42. Merge Two Lists

Use `+` operator or `extend` method to merge two lists:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
merged_list = list1 + list2  # Output: [1, 2, 3, 4, 5, 6]
# or
list1.extend(list2)          # Output: [1, 2, 3, 4, 5, 6]
```

43. Inline `if` Statement

Use inline `if` for concise conditional expressions:

```
x = 10
result = "positive" if x > 0 else "non-positive"
```

44. `sum` with List Comprehension

Use `sum` with list comprehension to perform arithmetic operations on lists:

```
numbers = [1, 2, 3, 4, 5]
sum_squared = sum(x ** 2 for x in numbers)  # Output: 55
```

45. Pass Multiple Arguments

Pass multiple arguments to a function using the `*args` and `**kwargs` syntax:

```
def my_function(*args, **kwargs):
    for arg in args:
        print(arg)

my_function(1, 2, 3, 4)  # Output: 1 2 3 4

def my_function(**kwargs):
    for key, value in kwargs.items():
        print(key, value)

my_function(a=1, b=2, c=3)  # Output: a 1, b 2, c 3
```

46. `chr` and `ord`

Convert characters to their ASCII codes and vice versa:

```
char = 'A'
ascii_code = ord(char)  # Output: 65

ascii_code = 65
char = chr(ascii_code)  # Output: 'A'
```

47. `all` and `any`

Check if all or any elements in a sequence satisfy a condition:

```
numbers = [2, 4, 6, 7, 8]
any_even = any(x % 2 == 0 for x in numbers)
all_even = all(x % 2 == 0 for x in numbers)
```

48. `round`

Use `round` to round numbers to a specific number of decimal places:

```
number = 3.14159
rounded_number = round(number, 2)  # Output: 3.14
```

49. List Concatenation and Repetition

Use list concatenation (`+`) and repetition (`*`) for list operations:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
concatenated_list = list1 + list2  # Output: [1, 2, 3, 4, 5, 6]
repeated_list = list1 * 3  # Output: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

50. The `map` Function

Use `map` to apply a function to all elements of a list:

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x ** 2, numbers))
```

51. `max` and `min` with Key Function

Use `max` and `min` with a key function to find the element with the maximum or minimum value based on a specific attribute:

```
people = [
    {'name': 'Alice', 'age': 30},
    {'name': 'Bob', 'age': 25},
    {'name': 'Charlie', 'age': 35}
]
oldest_person = max(people, key=lambda x: x['age'])
```

By: Waleed Mousa

```
youngest_person = min(people, key=lambda x: x['age'])
```

52. sorted with Key Function

Use sorted with a key function to sort a list based on a specific attribute:

```
people = [
    {'name': 'Alice', 'age': 30},
    {'name': 'Bob', 'age': 25},
    {'name': 'Charlie', 'age': 35}
]
sorted_by_age = sorted(people, key=lambda x: x['age'])
```

53. The itertools Module

The itertools module provides a collection of tools for working with iterators:

```
import itertools

# Cartesian Product
colors = ['red', 'blue']
sizes = ['small', 'large']
product = list(itertools.product(colors, sizes))
# Output: [('red', 'small'), ('red', 'large'), ('blue', 'small'),
('blue', 'large')]

# Permutations
letters = ['a', 'b', 'c']
perms = list(itertools.permutations(letters, 2))
# Output: [('a', 'b'), ('a', 'c'), ('b', 'a'), ('b', 'c'), ('c', 'a'),
('c', 'b')]

# Combinations
letters = ['a', 'b', 'c']
combinations = list(itertools.combinations(letters, 2))
# Output: [('a', 'b'), ('a', 'c'), ('b', 'c')]
```

54. The `random` Module

The `random` module provides functions for generating random numbers:

```
import random

# Random Integer
random_number = random.randint(1, 10)

# Random Float
random_float = random.random()  # Range: 0.0 <= random_float < 1.0

# Random Choice
choices = ['apple', 'banana', 'cherry']
random_choice = random.choice(choices)
```

55. The `datetime` Module

The `datetime` module provides classes for working with dates and times:

```
import datetime

# Get Current Date and Time
current_datetime = datetime.datetime.now()

# Format Date
formatted_date = current_datetime.strftime('%Y-%m-%d')

# Calculate Time Difference
date1 = datetime.datetime(2023, 1, 1)
date2 = datetime.datetime(2023, 12, 31)
time_difference = date2 - date1
```

56. The `collections.defaultdict` for Counting Elements

Use `defaultdict` to count occurrences of elements in a list:

```
from collections import defaultdict

numbers = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
number_counts = defaultdict(int)
for num in numbers:
    number_counts[num] += 1
# Output: defaultdict(<class 'int'>, {1: 1, 2: 2, 3: 3, 4: 4})
```

57. The `collections.Counter`

Use `Counter` to count occurrences of elements in a list:

```
from collections import Counter

numbers = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
number_counts = Counter(numbers)
# Output: Counter({4: 4, 3: 3, 2: 2, 1: 1})
```

58. The `json` Module

The `json` module allows you to work with JSON data:

```
import json

# Convert Dictionary to JSON
data = {'name': 'Alice', 'age': 30}
json_data = json.dumps(data)

# Convert JSON to Dictionary
json_data = '{"name": "Alice", "age": 30}'
data = json.loads(json_data)
```

59. The `sys` Module

The `sys` module provides access to some variables used or maintained by the Python interpreter:

```
import sys

# Command-Line Arguments
script_name = sys.argv[0]
arguments = sys.argv[1:]

# Maximum Recursion Depth
max_recursion_depth = sys.getrecursionlimit()

# Current Platform
current_platform = sys.platform
```

60. The `os` Module

The `os` module provides a way of using operating system-dependent functionality:

```
import os

# Current Working Directory
current_directory = os.getcwd()

# Change Directory
os.chdir('/path/to/directory')

# List Directory Contents
directory_contents = os.listdir()

# Create Directory
os.makedirs('/path/to/new_directory')

# Remove Directory
os.rmdir('/path/to/empty_directory')
```

61. The `pickle` Module

The `pickle` module allows you to serialize Python objects:

```
import pickle

# Serialize Object to File
data = {'name': 'Alice', 'age': 30}
with open('data.pkl', 'wb') as file:
    pickle.dump(data, file)

# Deserialize Object from File
with open('data.pkl', 'rb') as file:
    loaded_data = pickle.load(file)
```

62. List Slicing

Use list slicing to extract sublists from a list:

```
numbers = [1, 2, 3, 4, 5]
sublist = numbers[1:4]  # Output: [2, 3, 4]
```

63. String Methods

Use built-in string methods for string manipulation:

```
text = "Hello, World!"

# Convert to Lowercase
lowercase_text = text.lower()  # Output: "hello, world!"

# Convert to Uppercase
uppercase_text = text.upper()  # Output: "HELLO, WORLD!"

# Capitalize First Letter
capitalized_text = text.capitalize()  # Output: "Hello, world!"

# Count Occurrences
count = text.count('l')  # Output: 3

# Replace Substrings
replaced_text = text.replace('World', 'Python')  # Output: "Hello, Python!"
```

64. The `str.join()` Method

Use `str.join()` to concatenate elements of a list into a single string:

```
words = ['Hello', 'World', '!']
sentence = ' '.join(words)  # Output: "Hello World !"
```

65. The `enumerate` Function

Use `enumerate` to get both the index and value of elements in a list:

```
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits):
    print(f"Index {index}: {fruit}")
```

66. The `open` Function with `with` Statement

Use the `with` statement to automatically handle file closing:

```
with open("file.txt", "r") as file:
    content = file.read()
# No need to manually close the file, it's automatically handled by the
'with' block.
```

67. `is` and `==` for Object Comparison

Use `is` to check if two variables reference the same object and `==` for value comparison:

```
list1 = [1, 2, 3]
list2 = [1, 2, 3]
if list1 is list2:
    print("Same object")
else:
    print("Different objects")
# Output: "Different objects"

if list1 == list2:
    print("Equal values")
else:
    print("Different values")
# Output: "Equal values"
```

By: Waleed Mousa

68. `while` Loop with `else` Statement

Use `else` with `while` loops to execute code when the loop condition becomes False:

```
count = 0
while count < 5:
    print(count)
    count += 1
else:
    print("Loop finished!")
# Output: "0 1 2 3 4 Loop finished!"
```

69. `for` Loop with `else` Statement

Use `else` with `for` loops to execute code when the loop completes without encountering a `break` statement:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    if fruit == "banana":
        break
else:
    print("No 'banana' found!")
# Output: "No 'banana' found!"
```

70. Dictionary Comprehensions

Use dictionary comprehensions for concise dictionary creation:

```
squared_dict = {x: x**2 for x in range(1, 11)}
```

71. Timeit Module for Code Timing

Use the `timeit` module to measure the execution time of code:

```
import timeit

def my_function():
    return sum(range(1000000))

execution_time = timeit.timeit(my_function, number=100)
print(f"Execution time: {execution_time} seconds")
```

By: Waleed Mousa

72. else with try and except

Use else with try and except to specify code that should be executed if no exceptions are raised:

```
try:
    result = 10 / 2
except ZeroDivisionError as e:
    print(f"Error: {e}")
else:
    print(f"Result: {result}")
```

73. zip and Unpacking for Parallel Iteration

Use zip and unpacking to iterate over multiple lists in parallel:

```
names = ["Alice", "Bob", "Charlie"]
scores = [85, 92, 78]
for name, score in zip(names, scores):
    print(f"{name}: {score}")
```

74. try with finally

Use finally to specify cleanup code that will always be executed, whether an exception occurs or not:

```
try:
    # Code that may raise an exception
    result = 10 / 2
finally:
    # Cleanup code that will always be executed
    print("Cleanup")
```

75. The isinstance Function for Type Checking

Use isinstance for type checking instead of using type:

```
if isinstance(value, int):
    # Do something with an integer value
```

76. List Comprehension with Condition

Use list comprehension with a condition to filter elements in a list:

```
numbers = [1, 2, 3, 4, 5]
even_numbers = [x for x in numbers if x % 2 == 0]
```

77. Dictionary Get with Default Value

Use get to access dictionary values with a default value if the key is not present:

```
my_dict = {'a': 1, 'b': 2}
value = my_dict.get('c', 0)  # Output: 0
```

78. The del Statement

Use del to remove elements from a list or delete variables:

```
my_list = [1, 2, 3, 4, 5]
del my_list[2]  # Remove the element at index 2

x = 10
del x  # Delete the variable 'x'
```

79. enumerate with Start Index

Use enumerate with a start index to specify the initial value of the index:

```
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits, start=1):
    print(f"Index {index}: {fruit}")
```

80. The `re` Module for Regular Expressions

The `re` module allows you to work with regular expressions:

```
import re

text = "Hello, World! My name is Alice."
pattern = r"Hello, (.*?)[.!]"
match = re.search(pattern, text)
if match:
    name = match.group(1)
    print(f"Name: {name}")
# Output: "Name: World"
```

81. The `sys.exit()` Function

Use `sys.exit()` to exit the Python script:

```
import sys

def my_function():
    # Some code
    if some_condition:
        sys.exit()

    # Other code
```

82. The `inspect` Module

The `inspect` module provides functions for inspecting live objects:

```
import inspect

def my_function(a, b=10, *args, c=20, **kwargs):
    pass

# Get Function Arguments
argspec = inspect.getfullargspec(my_function)
# Output: FullArgSpec(args=['a', 'b'], varargs='args', varkw='kwargs',
defaults=(10,), kwonlyargs=['c'], kwonlydefaults={'c': 20}, annotations={})
```

```
# Get Function Signature
signature = inspect.signature(my_function)
# Output: <Signature (a, b=10, *args, c=20, **kwargs)>

# Get Source Code
source_code = inspect.getsource(my_function)
```

83. The `open` Function with Different Modes

Use the `open` function with different modes for different file operations:

```
# Read Mode (default)
with open("file.txt", "r") as file:
    content = file.read()

# Write Mode (create a new file or overwrite an existing one)
with open("file.txt", "w") as file:
    file.write("Hello, World!")

# Append Mode (open for writing, but append to the end of the file)
with open("file.txt", "a") as file:
    file.write("\nGoodbye, World!")

# Binary Mode (read or write binary data)
with open("file.bin", "wb") as file:
    file.write(b'\x01\x02\x03')
```

84. String Formatting with `f-string`

Use f-strings (Python 3.6+) for string formatting:

```
name = "Alice"
age = 30
print(f"My name is {name} and I am {age} years old.")
# Output: "My name is Alice and I am 30 years old."
```

85. `__name__` and `__main__`

Use `__name__` and `__main__` to create reusable modules:

```
# my_module.py
def my_function():
    print("Hello from my_function!")

if __name__ == "__main__":
    # Executed when the script is run directly
    my_function()
```

86. `range` Function

Use the `range` function to generate a sequence of numbers:

```
numbers = list(range(1, 11))
# Output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

87. `pass` Statement

Use the `pass` statement as a placeholder for code that will be implemented later:

```
def my_function():
    # TODO: Implement this function
    pass
```

88. The `try`, `except`, and `finally` Block

Use `try`, `except`, and `finally` for exception handling:

```
try:
    # Code that may raise an exception
    result = 10 / 0
except ZeroDivisionError as e:
    # Exception handling
    print(f"Error: {e}")
finally:
    # Cleanup code that will always be executed
    print("Cleanup")
```

89. Namedtuples for Simple Classes

Use `namedtuple` to create simple classes with named fields:

```
from collections import namedtuple

Person = namedtuple("Person", ["name", "age"])
person = Person("Alice", 30)
print(person.name)  # Output: "Alice"
print(person.age)   # Output: 30
```

90. `sorted` Function with Custom Key

Use `sorted` with a custom key function to sort a list based on a specific attribute:

```
people = [
    {'name': 'Alice', 'age': 30},
    {'name': 'Bob', 'age': 25},
    {'name': 'Charlie', 'age': 35}
]
sorted_by_age = sorted(people, key=lambda x: x['age'])
```

91. Dictionary `setdefault` Method

Use `setdefault` to set a default value for a missing key in a dictionary:

```
my_dict = {'a': 1, 'b': 2}
value = my_dict.setdefault('c', 0)  # Output: 0
```

92. Time Formatting with `datetime`

Use `datetime` for time formatting:

```
import datetime

now = datetime.datetime.now()

# Format as ISO 8601
formatted_time = now.isoformat()  # Output:
"2023-08-02T12:34:56.789012"
```

By: Waleed Mousa

```
# Custom Format
formatted_time = now.strftime("%Y-%m-%d %H:%M:%S.%f")  # Output:
"2023-08-02 12:34:56.789012"
```

93. The os Module for File Operations

The os module provides functions for file-related operations:

```
import os

# Get Current Working Directory
current_directory = os.getcwd()

# List Files in a Directory
files = os.listdir('/path/to/directory')

# Check if a File or Directory Exists
exists = os.path.exists('/path/to/file_or_directory')

# Create a Directory
os.makedirs('/path/to/new_directory')

# Remove a File
os.remove('/path/to/file')

# Rename a File
os.rename('/path/to/old_file', '/path/to/new_file')

# Get File Size
file_size = os.path.getsize('/path/to/file')
```

94. in Operator for Membership Checking

Check if an element exists in a list or dictionary:

```
numbers = [1, 2, 3, 4, 5]
if 3 in numbers:
    print("3 is in the list.")

my_dict = {'a': 1, 'b': 2}
if 'b' in my_dict:
```

```
    print("Key 'b' exists in the dictionary.")
```

95. The `time` Module for Time Operations

The `time` module provides functions for working with time:

```
import time

# Get Current Time in Seconds
current_time = time.time()

# Sleep for a Specified Time
time.sleep(3)  # Sleep for 3 seconds
```

96. `assert` Statement

Use `assert` for debugging and testing:

```
def my_function(x):
    assert x > 0, "x must be positive"
    # Rest of the code
```

97. The `random` Module for Random Numbers

The `random` module provides functions for generating random numbers:

```
import random

# Random Integer
random_number = random.randint(1, 10)

# Random Float
random_float = random.random()  # Range: 0.0 <= random_float < 1.0

# Random Choice
choices = ['apple', 'banana', 'cherry']
random_choice = random.choice(choices)
```

98. The `os.path` Module for Path Operations

The `os.path` module provides functions for path-related operations:

```
import os.path

# Check if a Path Exists
exists = os.path.exists('/path/to/file_or_directory')

# Get the Absolute Path
absolute_path = os.path.abspath('file.txt')

# Join Paths
path = os.path.join('/path/to', 'file.txt')

# Get the Base Name of a Path
base_name = os.path.basename('/path/to/file.txt')
```

99. `filter` Function for Filtering Elements

Use `filter` to filter elements in a list based on a condition:

```
numbers = [1, 2, 3, 4, 5]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
```

100. `lambda` Functions

Use `lambda` functions for simple, anonymous functions:

```
add = lambda x, y: x + y
result = add(3, 5)  # Output: 8
```

101. The `functools.partial` Function

Use `functools.partial` to create functions with fixed arguments:

```
import functools

def power(base, exponent):
    return base ** exponent

square = functools.partial(power, exponent=2)
cube = functools.partial(power, exponent=3)

print(square(5))  # Output: 25
print(cube(5))    # Output: 125
```