Python Quick Reference Guide

Python Quick Reference Guide

Operators: Special symbols used to perform operations on variables and values.

1. Arithmetic Operators - Perform basic arithmetic:

```
+, -, *, /, %, //, **
a = 5 + 2  # 7
b = 5 ** 2  # 25
```

2. Comparison Operators - Compare values:

```
==, !=, >, <, >=, <=
a = 5
b = 3
print(a > b)  # True
```

3. Assignment Operators - Assign and modify values:

```
=, +=, -=, *=, /=, //=, **=, %=
x = 10
x += 5  # x becomes 15
```

4. Logical Operators - Combine conditional statements:

```
and, or, not
x = True
y = False
print(x and y)  # False
```

5. Bitwise Operators - Operate on binary numbers:

```
&, |, ^, ~, <<, >>
a = 5 & 3  # 1
```

6. Identity Operators - Compare memory locations:

```
is, is not
x = [1, 2]
```

```
y = x
print(x is y)  # True
```

7. Membership Operators - Test for membership:

in, not in

```
x = [1, 2, 3]
print(2 in x)  # True
```

Loops: Used to execute a block of code repeatedly.

- for: Iterates over a sequence.

```
for i in range(3):
    print(i)
```

- while: Loops while a condition is true.

```
x = 0
while x < 3:
    print(x)
    x += 1
```

Break, Continue, Pass: Used for loop control.

- break: Exits the loop.

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

- continue: Skips to the next iteration.

```
for i in range(5):
    if i == 3:
        continue
    print(i)
```

- pass: Placeholder for future code.

```
for i in range(5):
    pass  # does nothing
```

Conditional Statements: Used to perform actions based on conditions.

- if, if-else, nested if.

```
x = 10
if x > 5:
    print("Greater")
else:
    print("Lesser")
```

Error Handling: Used to manage exceptions.

```
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
```

Built-in Functions:

```
len("hello")         # 5
print(type(10))      # <class 'int'>
list(range(3))       # [0, 1, 2]
name = input("Name: ")
print("Hello", name) # Output greeting
sum([1, 2, 3])       # 6
min([1, 2, 3])       # 1
max([1, 2, 3])       # 3
sorted([3, 1, 2])    # [1, 2, 3]
```

Higher-Order Functions:

- map(): Applies a function to all items.

```
list(map(lambda x: x*2, [1, 2, 3]))  # [2, 4, 6]
```

- filter(): Filters items based on a condition.

```
list(filter(lambda x: x%2 == 0, [1, 2, 3]))  # [2]
```

- zip(): Combines multiple iterables.

```
list(zip([1, 2], ['a', 'b']))  # [(1, 'a'), (2, 'b')]
```

```
- enumerate(): Adds counter to iterable.
list(enumerate(['a', 'b']))  # [(0, 'a'), (1, 'b')]

Data Types and Methods:

1. String: Sequence of characters.
s = "hello"
s.upper()       # 'HELLO'
s.lower()       # 'hello'
s.strip()       # removes whitespace
s.replace('l','x')  # 'hexxo'
s.split('e')    # ['h', 'llo']

2. List: Ordered, mutable collection.
lst = [1, 2, 3]
lst.append(4)    # [1, 2, 3, 4]
lst.pop()        # 4
lst.remove(2)    # [1, 3]
lst.sort()       # sorts the list

3. Set: Unordered, no duplicates.
s = {1, 2, 3}
s.add(4)
s.remove(2)
s.union({5})     # {1, 3, 4, 5}
s.intersection({3, 5})  # {3}

4. Tuple: Ordered, immutable collection.
t = (1, 2, 3)
t.count(2)       # 1
t.index(3)       # 2

5. Dictionary: Key-value pairs.
d = {'a': 1, 'b': 2}
d.get('a')       # 1
```

```
d.keys()         # dict_keys(['a', 'b'])
d.values()       # dict_values([1, 2])
d.items()        # dict_items([('a', 1), ('b', 2)])
d.update({'c': 3})  # {'a': 1, 'b': 2, 'c': 3}
```

Functions: Block of reusable code.

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

Recursive Function: Function calling itself.

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n-1)
```

Lambda Function: Anonymous one-liner function.

```
square = lambda x: x**2
square(3)  # 9
```

List Comprehensions: Concise way to create lists.

```
[x*2 for x in range(5)]  # [0, 2, 4, 6, 8]
```

Dictionary Comprehensions: Create dictionary with expression.

```
{x: x**2 for x in range(3)}  # {0: 0, 1: 1, 2: 4}
```

Regular Expressions: Pattern matching in strings.

```
import re
re.findall(r'\d+', 'abc123xyz456')  # ['123', '456']
```

- match(), search(), findall(), sub()

File Handling: Read/write files.

```
with open('file.txt', 'r') as f:
    data = f.read()
```

Modules:

- os: Provides functions to interact with the OS.

```
import os
print(os.getcwd())
```

- sys: Access system-specific parameters.

```
import sys
print(sys.version)
```

- paramiko: Used for SSH2 protocol.

```
import paramiko
ssh = paramiko.SSHClient()
```

OOPS Concepts:

1. Inheritance: Mechanism where one class can acquire properties (methods and variables) of another class.

```
class Animal:
    def sound(self): print("Sound")
class Dog(Animal): pass
```

- Single: One base class and one derived class.
- Multiple: A class inherits from multiple base classes.
- Multilevel: Inheritance across more than two levels.

2. Data Abstraction: Hides internal implementation and shows only essential features. Helps to reduce complexity.

```
from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def area(self): pass
```

3. Data Encapsulation: Restricts direct access to some components of an object, protecting the internal state.

```
class Person:
    def __init__(self):
        self.__age = 30  # private variable
```

4. Polymorphism: Ability to present the same interface for different data types, promoting flexibility.

```
class Dog:
    def sound(self): print("Bark")
class Cat:
    def sound(self): print("Meow")
for animal in (Dog(), Cat()):
    animal.sound()
```

5. Generator: A function that yields one item at a time using yield keyword.
Generators are memory-efficient because they produce items one by one and don't store the whole sequence in memory.

```
def gen():
    yield 1
    yield 2
# Usage
for value in gen():
    print(value)
```

Advantage: Uses less memory compared to lists.

```
import sys

def gen():
    for i in range(1000):
        yield i

g = gen()
print(sys.getsizeof(g))  # Much smaller than a list
```

6. Iterator: Object with __iter__() and __next__() methods, used to iterate over collections.

```
it = iter([1, 2])
print(next(it))  # 1
print(next(it))  # 2
```