Python Cheat Sheet for Quick Basic Fundamentals

Welcome to your comprehensive quick reference guide for Python! This cheat sheet covers the fundamental concepts you need to get started with Python programming, enhanced with extended examples for better understanding and quick revision.

Table of Contents

Basic Syntax

Overview: Learn the foundational rules and structure of Python code, including comments, indentation, and case sensitivity.

- **Comments:**
  - **Single-line Comments:** Use `#` to add single-line comments.
  - **Multi-line Comments:** Use `''' '''` or `""" """` for multi-line comments or docstrings.

```
# This is a single-line comment

"""
This is a
multi-line comment
"""
```

- **Indentation:**
  - Python uses indentation (usually 4 spaces) to define code blocks.
  - Consistent indentation is crucial as Python relies on it for block definitions.

```
if True:
    print("Indented block")
```

```python
    if False:
        print("This won't print")
print("Outside the block")
```

- **Case Sensitivity:**
    - Python is case-sensitive. Variables like `Variable`, `variable`, and `VARIABLE` are distinct.

```python
Variable = 10
variable = 20
print(Variable)  # 10
print(variable)  # 20
```

- **Line Continuation:**
    - Use backslashes (`\`) or parentheses to continue lines.

```python
total = 1 + 2 + 3 + \
        4 + 5

total = (1 + 2 + 3 +
         4 + 5)
```

Variables & Data Types

Overview: Understand how to store and manipulate data using variables and different data types in Python.

- **Variables:**
    - Assign values using `=`.
    - Variable names must start with a letter or underscore and can contain letters, numbers, and underscores.
    - Python supports dynamic typing; no need to declare variable types explicitly.

```python
x = 5
name = "Alice"
is_active = True
pi = 3.14159
```

- **Data Types:**
    - **Numeric:** `int`, `float`, `complex`
    - **String:** Enclosed in quotes (`' '`, `" "`, `''' '''`, `""" """`)
    - **Boolean:** `True`, `False`
    - **NoneType:** `None`

```
integer = 10
floating = 10.5
complex_num = 3 + 4j
string = "Hello, World!"
boolean = False
nothing = None
```

- **Type Checking and Conversion:**

```
x = 10
print(type(x))  # <class 'int'>

y = "20"
print(int(y))   # 20
print(float(y)) # 20.0

z = 3.14
print(str(z))   # "3.14"
```

- **Type Casting:**
    - Convert between types using `int()`, `float()`, `str()`, `bool()`, etc.

```
a = "100"
b = int(a)      # 100
c = float(a)    # 100.0
d = bool(a)     # True (non-empty string)
e = bool("")    # False (empty string)
```

- **Type Hinting (Optional):**
    - Improve code readability and assist IDEs with type checking.

```
def add(a: int, b: int) -> int:
    return a + b
```

Operators

Overview: Utilize various operators to perform operations on variables and values, including arithmetic, comparison, logical, and more.

- **Arithmetic Operators:**
    - `+` (Addition), `-` (Subtraction), `*` (Multiplication), `/` (Division)

  - `//` (Floor Division), `%` (Modulus), `**` (Exponentiation)

  ```
  a = 10
  b = 3

  print(a + b)   # 13
  print(a - b)   # 7
  print(a * b)   # 30
  print(a / b)   # 3.333...
  print(a // b)  # 3
  print(a % b)   # 1
  print(a ** b)  # 1000
  ```

- **Comparison Operators:**
  - `==` (Equal), `!=` (Not Equal), `>` (Greater Than), `<` (Less Than)
  - `>=` (Greater Than or Equal To), `<=` (Less Than or Equal To)

  ```
  print(5 == 5)   # True
  print(5 != 3)   # True
  print(5 > 3)    # True
  print(5 < 3)    # False
  print(5 >= 5)   # True
  print(5 <= 4)   # False
  ```

- **Logical Operators:**
  - `and`, `or`, `not`

  ```
  a = True
  b = False

  print(a and b)  # False
  print(a or b)   # True
  print(not a)    # False
  ```

- **Assignment Operators:**
  - `=`, `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`

  ```
  x = 5
  x += 3  # x = x + 3 → 8
  x -= 2  # x = x - 2 → 6
  x *= 4  # x = x * 4 → 24
  x /= 3  # x = x / 3 → 8.0
  x //= 2 # x = x // 2 → 4.0
  ```

```
x %= 3  # x = x % 3 → 1.0
x **= 2 # x = x ** 2 → 1.0
```

- **Bitwise Operators:**
  - `&` (AND), `|` (OR), `^` (XOR), `~` (NOT)
  - `<<` (Left Shift), `>>` (Right Shift)

```
a = 60  # 0011 1100
b = 13  # 0000 1101

print(a & b)  # 12 (0000 1100)
print(a | b)  # 61 (0011 1101)
print(a ^ b)  # 49 (0011 0001)
print(~a)     # -61 (in two's complement)
print(a << 2) # 240 (1111 0000)
print(a >> 2) # 15 (0000 1111)
```

- **Membership Operators:**
  - `in`, `not in`

```
fruits = ["apple", "banana", "cherry"]

print("banana" in fruits)     # True
print("orange" not in fruits) # True
```

- **Identity Operators:**
  - `is`, `is not`

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

print(a is b)      # True
print(a is c)      # False
print(a == c)      # True
print(a is not c)  # True
```

Control Structures

Overview: Control the flow of your programs using conditional statements and loops to execute code blocks based on conditions or repeatedly.

- **If Statement:**
    - Execute code based on conditions.

```
age = 20

if age >= 18:
    print("Adult")
elif age >= 13:
    print("Teenager")
else:
    print("Child")
```

- **Nested If Statements:**

```
num = 15

if num > 0:
    if num % 2 == 0:
        print("Positive even number")
    else:
        print("Positive odd number")
else:
    print("Non-positive number")
```

- **For Loops:**
    - Iterate over a sequence (like a list, tuple, string) or use `range()`.

```
# Iterating over a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)

# Using range()
for i in range(5):
    print(i)  # 0 to 4
```

- **While Loops:**
    - Repeat as long as a condition is true.

```
count = 0
while count < 5:
    print(count)
    count += 1
```

- **Break and Continue:**
    - `break`: Exit the loop.
    - `continue`: Skip to the next iteration.

```python
for i in range(10):
    if i == 5:
        break  # Exit loop when i is 5
    if i % 2 == 0:
        continue  # Skip even numbers
    print(i)  # Prints 1, 3
```

- **Else Clause with Loops:**
    - Executes after the loop finishes normally (no `break`).

```python
for i in range(3):
    print(i)
else:
    print("Loop completed without break")

for i in range(3):
    if i == 1:
        break
    print(i)
else:
    print("This won't print because of break")
```

- **Pass Statement:**
    - Placeholder for future code.

```python
def function():
    pass  # To be implemented later
```

Functions

Overview: Encapsulate reusable code blocks with functions, including defining, calling, and using advanced features like default parameters and lambda functions.

- **Defining Functions:**

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

```python
print(greet("Alice"))  # Output: Hello, Alice!
```

- **Calling Functions:**

```python
def add(a, b):
    return a + b

result = add(3, 5)
print(result)  # 8
```

- **Default Parameters:**
    - Parameters with default values.

```python
def greet(name="World"):
    return f"Hello, {name}!"

print(greet())         # Hello, World!
print(greet("Alice"))  # Hello, Alice!
```

- **Keyword Arguments:**
    - Pass arguments by name.

```python
def describe_pet(animal_type, pet_name):
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(animal_type="hamster", pet_name="harry")
```

- **Variable-length Arguments:**
    - *args (tuple) and **kwargs (dictionary).

```python
def make_pizza(size, *toppings):
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza(12, "pepperoni", "mushrooms", "green peppers")

def build_profile(first, last, **user_info):
    profile = {}
    profile['first_name'] = first
    profile['last_name'] = last
    for key, value in user_info.items():
        profile[key] = value
    return profile

user_profile = build_profile("albert", "einstein", location="princeton",
field="physics")
print(user_profile)
```

- **Lambda Functions:**
  - Anonymous, single-expression functions.

```
add = lambda a, b: a + b
print(add(2, 3))  # 5

# Sorting with lambda
points = [(2, 3), (1, 2), (4, 1)]
points.sort(key=lambda point: point[1])
print(points)  # [(4, 1), (1, 2), (2, 3)]
```

- **Docstrings:**
  - Documentation for functions using triple quotes.

```
def multiply(a, b):
    """
    Multiply two numbers and return the result.

    Parameters:
    a (int or float): The first number.
    b (int or float): The second number.

    Returns:
    int or float: The product of a and b.
    """
    return a * b

print(multiply(3, 4))  # 12
print(multiply.__doc__)
```

- **Recursion:**
  - Functions calling themselves.

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

print(factorial(5))  # 120
```

Data Structures

Overview: Manage and organize data efficiently using Python's built-in data structures like lists, tuples, dictionaries, and sets.

- **Lists:**
    - Ordered, mutable collections.
    - Support duplicate elements.

```
fruits = ["apple", "banana", "cherry"]
fruits.append("date")         # Add to end
fruits.insert(1, "blueberry")# Insert at index 1
print(fruits)                 # ['apple', 'blueberry', 'banana', 'cherry',
'date']

print(fruits[2])              # banana
print(fruits[-1])             # date

fruits.remove("banana")       # Remove by value
print(fruits)

popped = fruits.pop(2)        # Remove and return by index
print(popped)
print(fruits)

fruits.sort()                 # Sort alphabetically
print(fruits)

fruits.reverse()              # Reverse the list
print(fruits)

print(len(fruits))            # Number of elements
```

- **Tuples:**
    - Ordered, immutable collections.
    - Use parentheses `()` or no brackets.

```
coordinates = (10, 20)
print(coordinates[0])  # 10
print(coordinates[1])  # 20
```

```python
# Single-element tuple
single = (5,)
print(type(single))    # <class 'tuple'>

# Immutability
try:
    coordinates[0] = 15
except TypeError as e:
    print(e)  # 'tuple' object does not support item assignment
```

- **Dictionaries:**
  - Unordered (as of Python 3.7, insertion ordered), mutable collections of key-value pairs.

```python
person = {"name": "Alice", "age": 25, "city": "New York"}
print(person["name"])    # Alice
print(person.get("age")) # 25

person["age"] = 26         # Update value
person["email"] = "alice@example.com" # Add new key-value pair
print(person)

# Iterate over keys, values, items
for key in person:
    print(key)

for value in person.values():
    print(value)

for key, value in person.items():
    print(f"{key}: {value}")

# Remove items
removed = person.pop("city")
print(removed)
print(person)

person.clear()
print(person)  # {}
```

- **Sets:**
  - Unordered collections of unique elements.

```python
unique_numbers = {1, 2, 3, 2, 4, 3}
print(unique_numbers)  # {1, 2, 3, 4}

unique_numbers.add(5)
unique_numbers.remove(2)
print(unique_numbers)
```

```python
# Set operations
set_a = {1, 2, 3}
set_b = {3, 4, 5}

print(set_a.union(set_b))    # {1, 2, 3, 4, 5}
print(set_a.intersection(set_b)) # {3}
print(set_a.difference(set_b))   # {1, 2}
print(set_a.symmetric_difference(set_b)) # {1, 2, 4, 5}
```

- **List Comprehensions:**
  - Concise way to create lists.

```python
squares = [x**2 for x in range(10)]
print(squares)  # [0, 1, 4, 9, ..., 81]

even_squares = [x**2 for x in range(10) if x % 2 == 0]
print(even_squares)  # [0, 4, 16, 36, 64]
```

- **Dictionary Comprehensions:**

```python
squares_dict = {x: x**2 for x in range(5)}
print(squares_dict)  # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

- **Set Comprehensions:**

```python
unique_squares = {x**2 for x in range(-3, 4)}
print(unique_squares)  # {0, 1, 4, 9}
```

Input and Output

Overview: Handle user interaction by reading inputs and displaying outputs through the console.

- **Printing to Console:**
  - Use `print()` to display information.

```python
print("Hello, World!")
print("The answer is", 42)
print(f"The value of pi is approximately {3.14159}")
```

- **Formatting Output:**

- **Using f-strings (Python 3.6+):**

  ```
  name = "Alice"
  age = 25
  print(f"{name} is {age} years old.")
  ```

- **Using `str.format()`:**

  ```
  print("{} is {} years old.".format(name, age))
  ```

- **Using `%` Operator:**

  ```
  print("%s is %d years old." % (name, age))
  ```

- **Reading Input:**
  - Use `input()` to get user input from the console.

```
name = input("Enter your name: ")
print(f"Hello, {name}!")

age = int(input("Enter your age: "))
print(f"You are {age} years old.")
```

- **Handling Input Errors:**

```
try:
    age = int(input("Enter your age: "))
except ValueError:
    print("Please enter a valid number.")
```

- **Reading from Files:**

```
# Writing to a file
with open("example.txt", "w") as file:
    file.write("Hello, World!\n")
    file.write("This is a file.\n")

# Reading from a file
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

```python
# Reading lines
with open("example.txt", "r") as file:
    for line in file:
        print(line.strip())
```

- **Writing to Files:**

```python
data = ["Line 1\n", "Line 2\n", "Line 3\n"]
with open("output.txt", "w") as file:
    file.writelines(data)
```

- **Appending to Files:**

```python
with open("output.txt", "a") as file:
    file.write("Additional line.\n")
```

Exception Handling

Overview: Manage errors gracefully using try-except blocks to handle exceptions and ensure your program doesn't crash unexpectedly.

- **Basic Try-Except:**

```python
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

- **Multiple Except Blocks:**

```python
try:
    num = int(input("Enter a number: "))
    inverse = 1 / num
except ValueError:
    print("That's not a valid number!")
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

- **Else Clause:**
  - Executes if no exceptions occur.

```python
try:
    num = int(input("Enter a number: "))
    inverse = 1 / num
except (ValueError, ZeroDivisionError) as e:
    print(f"Error: {e}")
else:
    print(f"The inverse is {inverse}")
```

- **Finally Clause:**
  - Executes regardless of exceptions.

```python
try:
    file = open("data.txt", "r")
    data = file.read()
except FileNotFoundError:
    print("File not found!")
finally:
    file.close()
    print("File closed.")
```

- **Raising Exceptions:**
  - Use `raise` to trigger exceptions manually.

```python
def check_positive(number):
    if number < 0:
        raise ValueError("Negative value not allowed")
    return number

try:
    check_positive(-5)
except ValueError as e:
    print(e)  # Negative value not allowed
```

- **Custom Exceptions:**

```python
class CustomError(Exception):
    pass

def trigger_error():
    raise CustomError("This is a custom error!")

try:
    trigger_error()
except CustomError as e:
    print(e)  # This is a custom error!
```

- **Handling Multiple Exceptions Together:**

```python
try:
    # Some code that may raise multiple exceptions
    pass
except (TypeError, ValueError) as e:
    print(f"Error: {e}")
```

Modules and Packages

Overview: Organize your code and reuse functionality by importing modules and packages, and learn how to install external packages.

- **Importing Modules:**

```python
import math
print(math.sqrt(16))  # 4.0
print(math.pi)        # 3.141592653589793
```

- **Import Specific Functions or Variables:**

```python
from math import sqrt, pi
print(sqrt(25))  # 5.0
print(pi)        # 3.141592653589793
```

- **Import with Aliases:**

```python
import math as m
print(m.sqrt(9))  # 3.0

from math import factorial as fact
print(fact(5))    # 120
```

- **Import All from a Module:**
    - Not recommended due to potential namespace conflicts.

```python
from math import *
print(sin(pi/2))  # 1.0
```

- **Creating Your Own Modules:**
  - Save functions, classes, variables in `.py` files and import them.

  ```
  # my_module.py
  def greet(name):
      return f"Hello, {name}!"

  # main.py
  import my_module
  print(my_module.greet("Bob"))
  ```

- **Packages:**
  - Directories containing multiple modules and an `__init__.py` file.

  ```
  my_package/
      __init__.py
      module1.py
      module2.py
  ```

  ```
  # main.py
  from my_package import module1, module2
  module1.function()
  module2.class_name()
  ```

- **Installing External Packages (using pip):**

  ```
  pip install package_name
  ```

  - **Example: Installing `requests` library**

    ```
    pip install requests
    ```

  ```
  import requests
  response = requests.get("https://api.github.com")
  print(response.status_code)
  ```

- **Upgrading Packages:**

```
pip install --upgrade package_name
```

- **Uninstalling Packages:**

```
pip uninstall package_name
```

- **Listing Installed Packages:**

```
pip list
```

- **Virtual Environments:**
    - Isolate project dependencies using `venv`.

```
python -m venv myenv
source myenv/bin/activate  # On Unix or MacOS
myenv\Scripts\activate     # On Windows
pip install package_name
```

- **Using `__name__ == "__main__"`:**
    - Allow modules to be run as scripts or imported without executing certain code.

```
# my_module.py
def main():
    print("Module is being run directly")

if __name__ == "__main__":
    main()
```

Common Built-in Functions

Overview: Utilize Python's built-in functions to perform common tasks efficiently, such as iterating, type conversion, and more.

- **Range:**

```
for i in range(5):
    print(i)  # 0 to 4

# Specifying start and step
```

```
for i in range(2, 10, 2):
    print(i)  # 2, 4, 6, 8
```

- **len():**

```
print(len("Hello"))        # 5
print(len([1, 2, 3]))      # 3
print(len({'a':1, 'b':2})) # 2
```

- **type():**

```
print(type(123))        # <class 'int'>
print(type(3.14))       # <class 'float'>
print(type("Hello"))    # <class 'str'>
print(type([1, 2, 3]))  # <class 'list'>
```

- **str(), int(), float():**

```
num = 100
print(str(num))    # "100"

s = "10"
print(int(s) + 5)  # 15

f = "3.14"
print(float(f) * 2) # 6.28
```

- **print():**
  - Additional parameters:
    - `sep`: Separator between values (default is space).
    - `end`: What to print at the end (default is newline).

```
print("Hello", "World", sep="-")  # Hello-World
print("Hello", end=" ")           # Hello
print("World!")                    # World!
```

- **input():**

```
user_input = input("Enter something: ")
print(f"You entered: {user_input}")
```

- **enumerate():**
    - Iterate with index.

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

- **zip():**
    - Combine multiple iterables.

```
names = ["Alice", "Bob", "Charlie"]
ages = [25, 30, 35]

for name, age in zip(names, ages):
    print(f"{name} is {age} years old.")
```

- **map():**
    - Apply a function to all items in an iterable.

```
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, numbers))
print(squared)  # [1, 4, 9, 16]
```

- **filter():**
    - Filter items in an iterable based on a function.

```
numbers = [1, 2, 3, 4, 5, 6]
even = list(filter(lambda x: x % 2 == 0, numbers))
print(even)  # [2, 4, 6]
```

- **sorted():**
    - Return a sorted list from an iterable.

```
numbers = [4, 2, 1, 3]
sorted_numbers = sorted(numbers)
print(sorted_numbers)  # [1, 2, 3, 4]

# Sort in reverse
```

```
sorted_numbers_desc = sorted(numbers, reverse=True)
print(sorted_numbers_desc)  # [4, 3, 2, 1]
```

- **sum():**

```
numbers = [1, 2, 3, 4, 5]
total = sum(numbers)
print(total)  # 15
```

- **min() and max():**

```
numbers = [1, 2, 3, 4, 5]
print(min(numbers))  # 1
print(max(numbers))  # 5
```

- **any() and all():**

```
numbers = [0, 1, 2]
print(any(numbers))  # True (at least one is True)
print(all(numbers))  # False (not all are True)

all_true = [True, True, True]
print(all(all_true))  # True
```

- **abs():**

```
print(abs(-5))    # 5
print(abs(3.14))  # 3.14
```

- **round():**

```
print(round(3.14159, 2))  # 3.14
print(round(2.71828))      # 3
```

- **sorted() vs sort():**
  - `sorted()` returns a new sorted list.
  - `sort()` sorts the list in place.

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

```
print(sorted_numbers)  # [1, 2, 3, 4]
print(numbers)         # [3, 1, 4, 2]

numbers.sort()
print(numbers)         # [1, 2, 3, 4]
```

Additional Tips

- **Practice Regularly:**
  - Consistent coding helps reinforce concepts and improves problem-solving skills.
  - Try solving small projects or coding challenges regularly.
- **Read Documentation:**
  - The official Python documentation is a valuable resource for understanding built-in functions, modules, and best practices.
  - Use docstrings and help functions to get information about functions and modules.

```
help(print)
print.__doc__
```

- **Use Meaningful Variable Names:**
  - Enhances code readability and maintainability.

```
# Good
total_price = 100

# Bad
tp = 100
```

- **Keep Code Simple:**
  - Focus on writing clear and understandable code.
  - Avoid unnecessary complexity; follow the Zen of Python by running `import this`.
- **Debugging:**
  - Use print statements or debugging tools like `pdb` to troubleshoot your code.

```
import pdb

def faulty_function(x):
    pdb.set_trace()
    return x / 0
```

```
faulty_function(5)
```

- **Version Control:**
  - Use Git to track changes and collaborate on projects.

```
git init
git add .
git commit -m "Initial commit"
```

- **PEP 8 - Style Guide:**
  - Follow Python's style guidelines for writing clean and consistent code.
  - Use tools like `flake8` or `black` to enforce style.

```
pip install flake8
flake8 your_script.py
```

- **Learn to Use Virtual Environments:**
  - Isolate project dependencies to avoid conflicts.

```
python -m venv env
source env/bin/activate   # On Unix or MacOS
env\Scripts\activate      # On Windows
```

- **Explore Python Standard Library:**
  - Python comes with a rich set of modules for various tasks (e.g., `datetime`, `os`, `sys`, `json`, `re`).

```
import datetime
now = datetime.datetime.now()
print(now)

import os
print(os.listdir('.'))
```

- **Use Integrated Development Environments (IDEs):**
  - Tools like PyCharm, VSCode, or Jupyter Notebooks can enhance productivity with features like code completion, debugging, and more.

Happy Coding!