Master 20/80: Python

Abstract

The "Master 20/80: Python" document provides a concise guide to mastering Python, structured into four key sections: a comprehensive list of **all keywords** that highlights their meanings and uses, an overview of **symbols** that details various operators and syntactical elements, a set of **common rules** for effective Python programming practices, and a **master example** that integrates all discussed keywords, symbols, and rules into a cohesive code demonstration, showcasing practical applications of the language.

Table of Contents

1. All keywords:

Python has a set of reserved words, known as **keywords**, that have specific meanings and uses within the language. Here's the full list of Python keywords (as of Python 3.10) and their typical uses:

1. Control Flow Keywords

- **if**: Starts a conditional statement.
- **elif**: Specifies an alternative condition in an if statement.
- **else**: Defines a block of code to run if none of the preceding if or elif conditions are true.
- **for**: Begins a for loop, used to iterate over sequences (lists, tuples, strings, etc.).
- **while**: Begins a while loop, which repeats as long as a condition is true.
- **break**: Exits the nearest enclosing loop immediately.
- **continue**: Skips the rest of the code inside a loop for the current iteration, and continues with the next iteration.
- **pass**: Acts as a placeholder; used when a statement is required syntactically but you don't want any code to execute.
- **match**: Introduces pattern matching (introduced in Python 3.10).
- **case**: Specifies cases for pattern matching within a match statement.

2. Exception Handling Keywords

- **try**: Begins a block of code that will be tested for exceptions.
- **except**: Defines code to execute if an exception is raised in the try block.
- **finally**: Defines a block of code that will execute no matter what, even if an exception is raised.
- **raise**: Triggers an exception manually.
- **assert**: Tests if a condition is true; raises an AssertionError if not.

3. Function and Class Definition Keywords

- **def**: Declares a function.
- **return**: Specifies the return value of a function.
- **yield**: Pauses a generator function and provides a value to the caller while retaining its state for the next call.
- **lambda**: Creates an anonymous function (a function without a name).
- **class**: Defines a new class.
- **del**: Deletes an object, variable, or item from a list or dictionary.

4. Variable Scope Keywords

- **global**: Declares a variable as global, allowing it to be accessed and modified outside the local scope.
- **nonlocal**: Declares a variable to refer to a variable in the nearest enclosing scope that is not global.

5. Logical and Boolean Keywords

- **True**: Represents the boolean value True.
- **False**: Represents the boolean value False.
- **None**: Represents a null value or "no value here".
- **and**: Logical AND operator.
- **or**: Logical OR operator.
- **not**: Logical NOT operator, used to invert a boolean value.
- **is**: Tests for object identity, checks if two references point to the same object.
- **in**: Checks for membership, typically used to see if an item exists in a sequence.

6. Asynchronous Programming Keywords

- **async**: Used to define asynchronous functions, which can use await for asynchronous operations.
- **await**: Pauses asynchronous functions until the awaited task is complete.

7. Importing Modules Keywords

- **import**: Imports a module into the current namespace.
- **from**: Imports specific attributes or functions from a module.
- **as**: Renames a module or attribute during import.

8. Specialized Keywords

- **with**: Simplifies exception handling by automatically managing resources (e.g., files).
- **async with**: Used with asynchronous context managers.
- **async for**: Used with asynchronous iterators.

These keywords have specific syntactic or semantic roles in Python and cannot be used as identifiers (variable names, function names, etc.). Let me know if you need examples or further details on any of them!

2. Symbols

Python uses various symbols and operators that have specific meanings. Here's a comprehensive list of Python symbols and their functionality:

1. Arithmetic Operators

- **+** : Addition (e.g., 5 + 3) or concatenation (e.g., "Hello" + " World").
- **-** : Subtraction (e.g., 5 - 3) or negation (e.g., -5).
- ***** : Multiplication (e.g., 5 * 3), or repetition for sequences (e.g., ["a"] * 3).
- **/** : Division (e.g., 5 / 3) – returns a floating-point result.
- **//** : Floor Division (e.g., 5 // 3) – returns the largest integer less than or equal to the division result.
- **%** : Modulo (e.g., 5 % 3) – returns the remainder of division.
- ****** : Exponentiation (e.g., 5 ** 3) – raises the first number to the power of the second.

2. Assignment Operators

- **=** : Assigns a value to a variable (e.g., x = 5).
- **+=** : Adds and assigns (e.g., x += 5 is equivalent to x = x + 5).
- **-=** : Subtracts and assigns (e.g., x -= 5).
- ***=** : Multiplies and assigns (e.g., x *= 5).
- **/=** : Divides and assigns (e.g., x /= 5).
- **//=** : Floor divides and assigns (e.g., x //= 5).
- **%=** : Takes the modulo and assigns (e.g., x %= 5).
- ****=** : Raises to the power and assigns (e.g., x **= 5).

3. Comparison Operators

- **==** : Equal to (e.g., 5 == 3 returns False).
- **!=** : Not equal to (e.g., 5 != 3 returns True).
- **>** : Greater than (e.g., 5 > 3 returns True).
- **<** : Less than (e.g., 5 < 3 returns False).
- **>=** : Greater than or equal to (e.g., 5 >= 3 returns True).
- **<=** : Less than or equal to (e.g., 5 <= 3 returns False).

4. Logical Operators

- **and** : Logical AND (e.g., True and False returns False).
- **or** : Logical OR (e.g., True or False returns True).
- **not** : Logical NOT (e.g., not True returns False).

5. Bitwise Operators

- **&** : Bitwise AND (e.g., 5 & 3 returns 1).
- **|** : Bitwise OR (e.g., 5 | 3 returns 7).
- **^** : Bitwise XOR (e.g., 5 ^ 3 returns 6).
- **~** : Bitwise NOT (e.g., ~5 returns -6).
- **<<** : Left shift (e.g., 5 << 1 returns 10).
- **>>** : Right shift (e.g., 5 >> 1 returns 2).

6. Membership and Identity Operators

- **in** : Checks membership in sequences (e.g., 'a' in 'apple' returns True).
- **not in** : Checks non-membership in sequences (e.g., 'b' not in 'apple' returns True).
- **is** : Checks object identity (e.g., a is b checks if a and b point to the same object).
- **is not** : Checks if two objects are not the same (e.g., a is not b).

7. Ternary Operator

- **<expression1> if <condition> else <expression2>** : Conditional expression (e.g., x = 5 if a > b else 10).

8. Special Symbols

- **:** : Used in function definitions, loops, and conditional statements (e.g., def func():).
- **,** : Separates elements in lists, function arguments, etc. (e.g., a, b = 1, 2).
- **.** : Accesses attributes and methods of an object (e.g., obj.method()).
- **;** : Allows multiple statements on a single line (e.g., a = 1; b = 2), though it's rarely used.
- **()** : Used for grouping, function calls, and defining tuples (e.g., my_func(), (1, 2)).
- **[]** : Used for indexing, slicing, and defining lists (e.g., my_list[0]).
- **{}** : Used for dictionary literals and set literals (e.g., {"key": "value"}).
- **{}** : Used in format strings (e.g., f"Value is {var}").
- **->** : Used in function annotations for return types (e.g., def func() -> int:).
- **@** : Used for decorators (e.g., @decorator).
- **=** : Assigns values to variables (e.g., x = 5).

9. Augmented Assignment Operators

- **+=** : Increment and assign (e.g., x += 5).
- **-=** : Decrement and assign (e.g., x -= 5).
- ***=** : Multiply and assign (e.g., x *= 5).
- **/=** : Divide and assign (e.g., x /= 5).
- **//=** : Floor divide and assign (e.g., x //= 5).
- **%=** : Modulo and assign (e.g., x %= 5).
- ****=** : Exponent and assign (e.g., x **= 5).
- **&=** : Bitwise AND and assign (e.g., x &= 5).
- **|=** : Bitwise OR and assign (e.g., x |= 5).
- **^=** : Bitwise XOR and assign (e.g., x ^= 5).
- **<<=** : Left shift and assign (e.g., x <<= 5).
- **>>=** : Right shift and assign (e.g., x >>= 5).

10. Other Symbols

- **#** : Comment symbol, used to add inline comments (e.g., # This is a comment).
- **""" or '''** : Multi-line string literals or comments (e.g., """This is a multi-line string""").
- **** : Escape character, used to introduce special characters (e.g., \n for newline, \t for tab).
- **** : Line continuation character (e.g., x = 1 + \ 2).
- **_** : Often used as a placeholder for values you want to ignore, or for variable names (e.g., for _ in range(5)).

These symbols and operators are foundational to Python's syntax, allowing you to define logic, manipulate data, and control the flow of a program. Let me know if you'd like more details or examples for any specific symbol!

3. Common Rules for python programming

Here are some of the most common and important rules in Python, covering naming conventions, indentation, statements, functions, and more:

1. Identifiers and Naming Rules

- **Identifiers** (e.g., variable names, function names, class names) must follow these rules:
  - Can contain letters (a-z, A-Z), digits (0-9), and underscores (_).
  - Must **not begin** with a digit. For example, 1variable is invalid, but variable1 is valid.
  - Python is case-sensitive, so myVariable and myvariable are considered different identifiers.
  - Cannot be a **keyword** (e.g., if, while, return).
  - Avoid using special characters (e.g., !, @, $) in variable names.
  - By convention:
    - Variable names and function names use snake_case (e.g., my_variable, process_data()).
    - Class names use CamelCase (e.g., MyClass, UserProfile).
    - Constants are usually written in ALL_CAPS (e.g., PI, MAX_SIZE).

2. Indentation

- Python uses **indentation** (whitespace) to define the scope of loops, functions, classes, and other structures, replacing the need for curly braces ({}).
- Consistent indentation is critical. Each block of code that belongs to a particular statement must be indented at the same level.
- Typically, 4 spaces are used for indentation (avoid using tabs and spaces together).

```
if condition:
    do_something()  # Indented by 4 spaces
```

- Incorrect indentation will raise an IndentationError.

3. Statements

- **Simple statements** (e.g., assignments, print statements) can go on a single line.
- x = 5
- print(x)
- **Multiple statements** can be written on the same line using a semicolon (;), though this is generally discouraged for readability:
- x = 5; y = 10; z = 15
- **Compound statements** (e.g., if, for, while, def, class) should be split across multiple lines using proper indentation to improve readability.

4. Comments

- **Single-line comments** begin with #, and everything after the # on that line is ignored by the interpreter.

```
# This is a comment
x = 5  # This is also a comment
```

- **Multi-line comments** can be written using triple quotes (''' or """), though they are technically multi-line strings and not strictly comments. They are typically used as **docstrings**:

```
"""
This is a multi-line comment.
It can span multiple lines.
"""
```

5. String Handling

- Strings in Python can be defined using either single (') or double (") quotes:
- 'Hello'
- "World"
- Triple quotes (''' or """) can be used for multi-line strings:

```
"""
This is a multi-line string.
It can span multiple lines.
"""
```

- Strings are immutable in Python, meaning you cannot modify them after they are created.

6. Type Declarations

- Python is **dynamically typed**, so variable types don't need to be explicitly declared. The type is inferred from the value assigned:

```
x = 5       # Integer
y = "Hello"  # String
z = 3.14    # Float
```

- However, **type hints** can be used for better code readability and static analysis:
- def add(a: int, b: int) -> int:
- return a + b

7. Functions

- Functions are defined using the def keyword followed by the function name, parameters, and a colon (:).
- The body of the function must be indented.
- Functions can have a return statement to return values.

```
def my_function(param1, param2):
  result = param1 + param2
  return result
```

8. Looping and Conditionals

- **if, elif, and else** are used for conditional statements. Parentheses around conditions are optional.

```
if x > 5:
    print("x is greater than 5")
elif x == 5:
    print("x is equal to 5")
else:
    print("x is less than 5")
```

- **for and while** loops are used for iteration.

```
for i in range(5):
    print(i)

while x < 10:
    print(x)
    x += 1
```

9. Built-in Functions and Methods

- Python has many built-in functions such as print(), len(), sum(), type(), etc.
- Use **methods** associated with data types. For example, strings have methods like .lower(), .upper(), .strip(), etc.

```
my_string = " Hello "
print(my_string.strip())  # Removes leading and trailing spaces
```

10. Exceptions

- Python uses **exceptions** to handle errors. A try block lets you test code for errors, and the except block handles them:

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

Exceptions can be raised using the raise keyword:

```
raise ValueError("Invalid value")
```

11. Imports and Modules

- Use the **import** statement to include external libraries and modules:

```
import math
print(math.sqrt(16))
```

You can also import specific parts of a module:

```
from math import sqrt
print(sqrt(16))
```

Use as to give modules an alias:

```
import numpy as np
```

12. Whitespace Sensitivity

- Python is **whitespace-sensitive**, meaning blank lines and spaces between statements are generally ignored, but indentation is crucial for defining code blocks.
- It is common practice to use blank lines to separate functions and classes to improve code readability.

13. Return Values

- A function without an explicit return statement will return None by default.

```
def func():
    pass

print(func())  # Prints None
```

14. Boolean Expressions

- Boolean values in Python are represented by True and False. Logical operators include and, or, and not:

```
if x > 5 and y < 10:
    print("Condition is true")
```

15. List, Tuple, and Dictionary Syntax

- Lists are created using square brackets ([]):

```
my_list = [1, 2, 3]
```

- Tuples are created using parentheses (()):

```
my_tuple = (1, 2, 3)
```

- Dictionaries are created using curly braces ({}):

```
my_dict = {"name": "Alice", "age": 25}
```

16. Docstrings

- Python allows the use of **docstrings** to document functions, classes, and modules. These are enclosed in triple quotes and appear right after the definition:

```
def my_function():
    """This is a docstring."""
    Pass
```

These are the most common rules you'll encounter when writing Python code. Following these rules ensures your code is clean, readable, and easy to maintain.

4. Master Example

A comprehensive Master Python code example that demonstrates the use of various keywords, symbols, and common rules:

```
# Importing the math module
import math

# Constants
PI = 3.14159  # Constant using ALL_CAPS

# Function to calculate the area of a circle
def area_of_circle(radius: float) -> float:
    """
    Calculate the area of a circle given its radius.

    Parameters:
        radius (float): The radius of the circle.

    Returns:
        float: The area of the circle.
    """
    return PI * (radius ** 2)

# Function to demonstrate different types of loops
def loop_examples():
    # For loop example
    for i in range(1, 6):  # Loop from 1 to 5
        print(f"Square of {i}: {i ** 2}")  # Arithmetic operator for squaring

    # While loop example
    count = 5
    while count > 0:
        print(f"Countdown: {count}")
        count -= 1  # Augmented assignment operator

# Function to demonstrate exception handling
def divide_numbers(num1: float, num2: float) -> float:
    """
    Divide two numbers, handling division by zero.

    Parameters:
        num1 (float): The numerator.
        num2 (float): The denominator.
```

```
    Returns:
        float: The result of the division.
    """
    try:
        return num1 / num2
    except ZeroDivisionError as e:
        print("Error: Cannot divide by zero.", e)
        return None

# Main function to execute the program
def main():
    print("Area of Circle with radius 5:", area_of_circle(5))  # Function call

    print("\nLoop Examples:")
    loop_examples()  # Function call

    print("\nDivision Examples:")
    result = divide_numbers(10, 2)
    if result is not None:
        print("10 divided by 2 is:", result)
    divide_numbers(10, 0)  # This will raise an exception

    # List, Tuple, and Dictionary Examples
    my_list = [1, 2, 3, 4, 5]
    my_tuple = (10, 20, 30)
    my_dict = {"name": "Alice", "age": 25}

    print("\nList:", my_list)
    print("Tuple:", my_tuple)
    print("Dictionary:", my_dict)

    # Membership and Identity Operators
    if 3 in my_list:  # Membership operator
        print("3 is in the list.")

    # Using 'is' operator
    a = [1, 2, 3]
    b = a
    if a is b:  # Identity operator
        print("a and b refer to the same object.")

# Entry point of the program
if __name__ == "__main__":
    main()  # Calling the main function
```