Python Basics

The Python interpreter is a program that reads and executes the code written in the Python programming language. Understanding how the interpreter reads and executes Python scripts is crucial for both new and experienced developers. Here's a concise summary of the process:

Contents

The Python Interpreter

Reading the Script When a Python script is run, the Python interpreter starts by reading the script file. It reads the script line by line, which allows for the sequential execution of code. If the script imports modules, the interpreter loads those modules at the time of import.

Compilation to Bytecode Before execution, Python compiles the source code into bytecode. Bytecode is a lower-level, platform-independent representation of the source code. This compilation step is transparent to the user and is done to speed up execution. The bytecode is then stored in `.pyc` files, and if the source code has not changed, Python can skip the compilation step and use the existing bytecode to save time in subsequent executions.

Execution of Bytecode by Python Virtual Machine (PVM) The bytecode is executed by the Python Virtual Machine (PVM), which is part of the Python interpreter. The PVM reads the bytecode instructions one by one and performs the specified operations. This includes arithmetic operations, control flow operations, and interactions with the Python object model (creating objects, invoking methods, etc.).

Dynamic Typing As Python is a dynamically typed language, the interpreter determines the type of each variable at runtime. This means that you can reassign variables to objects of different types, and the interpreter handles type checking dynamically as the code runs.

Error Handling If the interpreter encounters errors while reading, compiling, or executing the script, it halts execution and raises an exception. Syntax errors are detected during the reading phase, while runtime errors are caught during execution. The interpreter provides a traceback for unhandled exceptions, helping the developer to identify and fix the source of the error.

Interactive Mode Besides executing scripts, the Python interpreter can also run in interactive mode. In this mode, you can type Python code directly into the interpreter, which reads and executes the code immediately. This is useful for testing, debugging, and learning.

The process from reading a Python script to executing it involves several steps: reading the source code, compiling it to bytecode, and executing the bytecode with the Python Virtual Machine. This workflow enables Python to offer the simplicity and flexibility of an interpreted language with some of

the execution efficiency of compiled languages. Understanding this process can help developers write more efficient and effective Python code.

Assignments

Variable assignment in Python is a fundamental concept that involves allocating a specific piece of data to a named identifier, allowing for the data to be accessed and manipulated throughout your code. Here's an overview of how variable assignments are handled in Python:

```
# Basic variable assignment
x = 10
name = "Alice"

# Assigning multiple variables in one line
a, b, c = 5, 3.2, "Hello"

# Assigning the same value to multiple variables
x = y = z = 0
```

Key Points:

- Dynamic Typing: Python is dynamically typed, meaning you don't need to declare a variable's type ahead of time. The type is inferred at runtime based on the assigned value.

```
x = 4   # x is of type int
x = "S" # Now x is of type str
```

- Mutable vs Immutable Objects: The assignment also depends on the type of object (mutable or immutable). For immutable objects (like integers, floats, strings, tuples), every assignment creates a new object. For mutable objects (like lists, dictionaries, sets), variables can reference the same object, allowing for in-place modifications.

```
# Immutable example
x = 10
y = x
x = x + 1  # x now points to a new object

# Mutable example
a = [1, 2, 3]
b = a
b.append(4)  # Both 'a' and 'b' see the change
```

- Variable Names: Variables must start with a letter or an underscore, followed by letters, numbers, or underscores. They are case-sensitive.

```
# Valid variable names
_var = "valid"
varName2 = "also valid"

# Invalid variable names
2var = "invalid"  # SyntaxError
```

- Scope: The scope of a variable determines the part of the program where you can access it. Python supports global and local scopes, along with nonlocal scope for nested functions.

```
# Global variable (accessible for the whole script file)
x = "global"

def myFunc():
    # Local variable
```

```
    y = "local"
    print(x)  # Access global variable

myFunc()
```

- Global and Local Variables: Variables declared at the beginning of a script are global for all following lines of code. Conversely, variables declared inside a function are local and exist only within that function's scope.

```
x = "global"

def myFunc():
    # Local variable
    y = "local"
    x = "myFunc can modify x"
    return y

z = myFunc()
print(x)  # Outputs 'modified'
print(y)  # Can not access local variable
print(z)  # Can access z, because y is returned by myFunc() and stored into global variable z
```

Understanding how variable assignment works in Python is crucial for managing data stored in variables, especially when working with different data types and structures. This overview covers the basics, but there's much more to explore, especially concerning scope rules, memory management, and the idiosyncrasies of mutable vs immutable object handling.

Augmented Assignment

Augmented assignment in Python is a shorthand operation that combines a binary operation and an assignment operation into a single statement. It's a convenient way to apply an operation to a variable and then assign the result back to the same variable. Here's a brief overview of how augmented assignments work:

```
# Basic example
a = 5
a += 4  # Equivalent to a = a + 4
```

```
Operator    Equivalent
var += 1    var = var + 1
var -= 1    var = var - 1
var *= 1    var = var * 1
var /= 1    var = var / 1
var //= 1   var = var // 1
var %= 1    var = var % 1
var **= 1   var = var ** 1
```

Math Operators

Python supports a variety of mathematical operators out-of-the-box, that allow you to perform calculations on numbers.

```
Operators   Operation       Example
**      Exponent        2 ** 3 = 8
%       Modulus/Remainder   22 % 8 = 6
//      Integer division    22 // 8 = 2
/       Division        22 / 8 = 2.75
*       Multiplication      3 * 3 = 9
```

```
-       Subtraction     5 - 2 = 3
+       Addition        2 + 2 = 4
```

These operators are used in expressions to perform arithmetic operations and can be combined to form more complex calculations. Python follows the usual mathematical rules for order of operations (the PEMDAS rule: Parentheses, Exponents, Multiplication/Division, Addition/Subtraction), allowing for intuitive mathematical expressions.

Built-in Math Functions in Python

Python provides several built-in functions for performing mathematical operations. These functions allow for both basic and advanced mathematical operations without the need for external libraries.

abs(x)
: Returns the absolute value of a number. The argument can be an integer, a floating point number, or a complex number.

divmod(a, b)
: Takes two non-complex numbers as arguments and returns a pair of numbers consisting of their quotient and remainder when using integer division.

max(iterable, *[, key, default]) / max(arg1, arg2, *args[, key])
: Returns the largest item in an iterable or the largest of two or more arguments.

min(iterable, *[, key, default]) / min(arg1, arg2, *args[, key])
: Returns the smallest item in an iterable or the smallest of two or more arguments.

pow(x, y[, z])
: Returns `x` to the power of `y`; if `z` is present, returns `x` to the power of `y`, modulo `z` (computed more efficiently than `pow(x, y) % z`).

round(number[, ndigits])
: Rounds a number to a specified number of digits after the decimal point. If `ndigits` is omitted or `None`, it returns the nearest integer to its input.

sum(iterable[, start])
: Sums start and the items of an iterable from left to right and returns the total. `start` defaults to 0.

complex(real[, imag])
: Creates a complex number with the real part `real` and the imaginary part `imag`.

int(x[, base])
: Converts a number or string `x` to an integer, or returns 0 if no arguments are given. Optionally, the conversion can be done according to a specified `base`.

float(x)
: Converts a string or a number to floating point.

These functions cover a range of mathematical operations from arithmetic to rounding, power calculations, and working with complex numbers. For more specialized operations, the `math` and `cmath` modules provide additional tools.

Variable Naming

Variable naming in Python is an essential aspect of writing clear, readable code. Python variables are names that are used to refer to data that can be used and manipulated in a program. Here are some key points regarding variable naming in Python:

1. **Case Sensitivity:** Variable names in Python are case-sensitive. This means that "variable", "Variable", and "VARIABLE" would refer to three different variables.
2. **Naming Rules:**

```
    - Variable names must start with a letter (a-z, A-Z) or an underscore (_).
    - After the first character, variable names can contain letters, digits (0-9), or underscores (_).
    - Variable names cannot contain spaces or special characters, and they cannot be reserved words in Python
(like "if", "for", "return", etc.).
```

1. **Naming Conventions:**
    1. Snake Case: This is the preferred way to name variables in Python for normal variables and functions. It involves writing all words in lowercase and separating them with underscores, e.g., `my_variable`.
    2. Camel Case: While not typically used for variable names in Python, it starts with a lowercase letter and capitalizes the first letter of each subsequent word without spaces, e.g., `myVariable`. It's more common in other programming languages.
    3. CapWords (Pascal Case): Used for naming classes in Python, it capitalizes the first letter of each word, e.g., `MyClass`.
2. **Descriptive Names:** Choose meaningful and descriptive names that convey the purpose of the variable. This makes the code more readable and understandable, e.g., `user_age` instead of `a`.
3. **Short Names:** While variable names should be descriptive, they should also be concise to avoid overly long code lines, e.g., `num_users` instead of `number_of_users_registered`.
4. **Avoiding Confusion:** Avoid using characters that could be confused with one another, like the letter 'l' (lowercase L), the letter 'O' (uppercase o), or the number '0' (zero), which can make the code hard to read and maintain.

```python
# Example of good variable naming
user_name = "John Doe"
number_of_pages = 120
temperature_in_celsius = 22.5

# Examples of bad variable naming
a = "John Doe"  # Not descriptive
NumberOfPages = 120  # Not following snake case convention for variables
T = 22.5  # Not descriptive and potentially confusing
```

Choosing appropriate variable names is crucial for ensuring that code is easy to read, understand, and maintain.

Comments

Commenting in Python plays a crucial role in the development process, serving multiple purposes that enhance the readability, maintainability, and usability of code. Here's why commenting is important:

1. **Code Readability**: Comments help explain the purpose of specific blocks of code to developers, making it easier for someone new to the codebase (or for the original developer returning to the code after some time) to understand the functionality and logic without having to decipher the code line by line.
2. **Code Documentation**: Well-commented code serves as documentation for the project, explaining how various components work and how they are intended to be used. This is

particularly useful for complex algorithms, where the logic behind the code might not be immediately apparent.

3. **Debugging**: During development, comments can be used to temporarily disable portions of code (commenting out), making it easier to isolate and identify problematic sections. This is a quick way to perform troubleshooting or testing changes without permanently removing code snippets.
4. **Code Maintenance**: In the lifecycle of software development, codebases often undergo revisions and updates. Comments provide valuable context to future maintainers, explaining why certain decisions were made or highlighting potential areas for improvement or caution.
5. **Knowledge Sharing**: In collaborative projects, comments act as a channel for developers to communicate important information, assumptions, or limitations about the code, ensuring that knowledge is shared and not just kept in the mind of the individual developer.
6. **Best Practices and Standards**: In professional environments, adhering to commenting standards and best practices is part of writing clean, professional-grade code. It reflects well on the developer's attention to detail and commitment to quality.

Inline comment:

```
# This is a comment
```

Multiline comment:

```
# This is a
# multiline comment
```

Code with a comment:

```
a = 1  # initialization
```

Function docstring:

```
def foo():
    """
    This is a function docstring
    You can also use:
    ''' Function Docstring '''
    """
```

In summary, commenting is an essential practice in Python programming that aids in the understanding, use, and maintenance of code. Effective commenting can significantly impact the success and longevity of a project, making it a habit worth cultivating for any developer.

Basic Types

Python automatically determines the type of a variable based on the value assigned to it. The most basic variables types are integer, floats and strings. The **type()** function returns the type of a variable or object.

```
a = 6
type(a) #  returns <class 'int'>

b = 6.6
type(b) #  returns <class 'float'>
```

```
c = "6.666"
type(c) #  returns <class 'str'>
```

Type-casting

- Python automatically determines the type of a variable based on the value assigned to it.
- Type conversion is straightforward in Python; you can convert between int, float, and str using int(), float(), and str() functions, respectively.
- Python supports a wide range of operations for numerical data types (int and float) and offers numerous methods for string manipulation.

```
a = 6.666
type(a) #  returns <class 'float'>

a = str(a) # returns "6.666"
type(a) #  returns <class 'str'>

a = int(float(a)) # returns 6
type(a) #  returns <class 'int'>
```

Builtin Methods of Basic Types

Python's basic data types — str, int, and float—come equipped with a variety of builtin methods that allow you to perform common tasks related to each type. These methods provide a powerful set of tools for manipulating these data types without the need for external libraries.

Object Introspection

Using the dir() Function for Inspection: The dir() function is a powerful tool for introspection in Python. It returns a list of valid attributes and methods for any object. This means you can use dir() to inspect what operations are available for str, int, float, or any other object.

```
print(dir("Hello"))  # Lists all string methods
print(dir(10))  # Lists all integer methods
print(dir(10.5))  # Lists all float methods
```

Using dir() is especially helpful when you're exploring new libraries or working with complex objects and need to understand their capabilities without referring to external documentation.

Using Objects

Objects combine attributes (data) and methods (operations) and are a powerful programming concept. Dot-notation in Python is a fundamental concept used to access methods and attributes of objects. It allows you to invoke functionality specific to an object's type or class and to retrieve data associated with that object. Understanding and using dot-notation is crucial for effective Python programming, especially when working with complex data types or when creating and using custom classes.

Accessing Methods

Methods are functions that are associated with an object and can act on the data contained within that object. To call a method on an object, you use dot-notation by appending a period (.) followed by the method name to the object's name. If the method requires arguments, you include them within parentheses after the method name.

```
# Example of calling a string object's method
text = "hello world"
upper_text = text.upper()  # Calls the upper() method on text
print(upper_text)  # Outputs: "HELLO WORLD"
```

Accessing Attributes

Attributes are variables that hold data related to an object. Like methods, attributes are accessed using dot-notation, but you do not use parentheses because you are not calling a function but rather accessing a variable.

```
# Example of accessing an object's attribute
import datetime
now = datetime.datetime.now()  # Creates a datetime object representing the current moment
year = now.year  # Accesses the year attribute of the datetime object
print(year)  # Outputs the current year
```

Important Points

- Object-specific: The methods and attributes you can access using dot-notation depend on the object's type. For example, string objects have string-specific methods like .upper() or .strip(), while list objects have list-specific methods like .append() or .sort().
- Custom Objects: When you define your own classes in Python, you can also define methods and attributes that can be accessed using dot-notation. This is a key part of object-oriented programming in Python.
- Method Calls vs. Attribute Access: Remember to use parentheses when calling methods to differentiate them from attribute access. Failing to do so (if you're actually trying to call a method) will result in a reference to the method itself rather than invoking the method.

String methods

My apologies for the oversight. Let me provide the list of common and useful string methods in MediaWiki format:

- **`upper()`**: Converts all characters in the string to uppercase.
  - ```
    "hello".upper()
    ```
- **`lower()`**: Converts all characters in the string to lowercase.
  - ```
    "HELLO".lower()
    ```
- **`strip()`**: Removes any leading and trailing whitespaces from the string.

```
" hello ".strip()
```

- **`lstrip()`**: Removes leading (left side) whitespace from the string.

```
" hello".lstrip()
```

- **`rstrip()`**: Removes trailing (right side) whitespace from the string.

```
"hello ".rstrip()
```

- **`split(sep=None)`**: Splits the string into a list of substrings based on a separator `sep`. If `sep` is not specified or is `None`, any whitespace string is a separator.

```
"hello world".split()
```

- **`join(iterable)`**: Joins an iterable of strings into a single string separated by the string providing this method.

```
" ".join(["hello", "world"])
```

- **`replace(old, new[, count])`**: Replaces occurrences of a substring within the string with a new substring, optionally limiting the number of replacements with `count`.

```
"hello world".replace("world", "Python")
```

- **`find(sub[, start[, end]])`**: Returns the lowest index in the string where substring `sub` is found. Returns `-1` if the substring is not found.

```
"hello world".find("world")
```

- **`index(sub[, start[, end]])`**: Like `find()`, but raises a `ValueError` when the substring is not found.

```
"hello world".index("world")
```

- **`count(sub[, start[, end]])`**: Returns the number of occurrences of a substring in the string, optionally within the bounds of `start` and `end`.

```
"hello world".count("o")
```

- **`startswith(prefix[, start[, end]])`**: Returns `True` if the string starts with the specified `prefix`, otherwise returns `False`.
  - ```
    "hello world".startswith("hello")
    ```
- **`endswith(suffix[, start[, end]])`**: Returns `True` if the string ends with the specified `suffix`, otherwise returns `False`.
  - ```
    "hello world".endswith("world")
    ```
- **`capitalize()`**: Capitalizes the first letter of the string and lowers all other letters.
  - ```
    "hello world".capitalize()
    ```
- **`title()`**: Converts the first character of each word to uppercase and the rest to lowercase.
  - ```
    "hello world".title()
    ```
- **`isnumeric()`**: Returns `True` if all characters in the string are numeric, otherwise `False`.
  - ```
    "12345".isnumeric()
    ```
- **`isalpha()`**: Returns `True` if all characters in the string are alphabetic, otherwise `False`.
  - ```
    "hello".isalpha()
    ```
- **`isalnum()`**: Returns `True` if all characters in the string are alphanumeric (either alphabets or numbers), otherwise `False`.
  - ```
    "hello123".isalnum()
    ```

These methods are part of the string class in Python and are highly useful for various text processing and manipulation tasks.

Integer methods

In Python, integers (`int`) do not have as many specific methods as strings do because they are designed to represent whole numbers and interact with other data types through basic arithmetic operations and conversions. However, there are several functions and operations that are commonly used with integers, as well as methods available through Python's standard library that are applicable to integers. Here's an overview in MediaWiki format:

- **Conversion and Utility Functions**
    ```

- `**int(x)**`: Converts a number or string `x` to an integer, if possible.
- `**abs(x)**`: Returns the absolute value of a number. The argument may be an integer.
- `**pow(x, y)**`: Raises `x` to the power of `y`. Both `x` and `y` can be integers, and `pow(x, y, z)` can be used for `x` raised to the power of `y`, modulo `z`.
- `**bin(x)**`: Converts an integer to a binary string.
- `**hex(x)**`: Converts an integer to a hexadecimal string.
- `**oct(x)**`: Converts an integer to an octal string.

- **Mathematical Functions in the `math` Module**
  - The `math` module provides access to mathematical functions for floating-point numbers, but some functions are relevant for integers, especially for conversion or when the result of a function applied to an integer is intended to be used as an integer.
  - `**math.sqrt(x)**`: Calculates the square root of `x`. While this returns a floating point number, it's commonly used with integers.
  - `**math.factorial(x)**`: Returns the factorial of `x`, an integer, which is an exact integer.
  - `**math.gcd(x, y)**`: Calculates the Greatest Common Divisor of `x` and `y`. If both inputs are integers, the result is also an integer.

- **Bitwise Operations**
  - Bitwise operations are not methods but operators that perform operations on the binary representations of integers.
  - `**x | y**`: Bitwise OR of `x` and `y`.
  - `**x & y**`: Bitwise AND of `x` and `y`.
  - `**x ^ y**`: Bitwise XOR of `x` and `y`.
  - `**x << n**`: Left shift `x` by `n` bits.
  - `**x >> n**`: Right shift `x` by `n` bits.
  - `**~x**`: Bitwise NOT of `x`.

While the list above does not represent specific methods tied to the integer type like those available to strings, it highlights the key functions and operations that are most commonly used with integers in Python programming. These tools are essential for numerical calculations, data manipulation, and algorithm implementation involving integers.

Float methods

For floating-point numbers (`float`), Python provides a set of operations and functions rather than methods directly attached to the float type itself. These operations and functions are essential for performing arithmetic calculations, rounding, and conversions. Here's an overview of the most useful and common operations and functions for working with floats, formatted in MediaWiki format:

- **Arithmetic Operations**
  - Arithmetic operations with floats follow the same basic arithmetic rules but are used for calculations requiring fractional parts.

- **Conversion and Utility Functions**
  - `**float(x)**`: Converts a string or a number `x` to a floating-point number, if possible.
  - `**abs(x)**`: Returns the absolute value of a floating-point number.
  - `**round(x[, n])**`: Rounds a floating-point number `x` to `n` digits after the decimal point. If `n` is omitted, it rounds to the nearest integer.

- **Mathematical Functions in the `math` Module**
  - The `math` module provides a wide range of mathematical operations that can be performed on floats. Here are some of the most commonly used functions:
    - **`math.ceil(x)`**: Returns the smallest integer greater than or equal to `x`.
    - **`math.floor(x)`**: Returns the largest integer less than or equal to `x`.
    - **`math.sqrt(x)`**: Returns the square root of `x`.
    - **`math.exp(x)`**: Returns `e` raised to the power of `x`.
    - **`math.log(x[, base])`**: Returns the logarithm of `x` to the given `base`. If the base is not specified, returns the natural logarithm.
    - **`math.pow(x, y)`**: Returns `x` raised to the power of `y`.
    - **`math.sin(x)`**, **`math.cos(x)`**, **`math.tan(x)`**: Trigonometric functions that return the sine, cosine, and tangent of `x`, respectively.
- **Type Conversion**
  - While not specific methods, it's common to convert floats to other types, especially to `int` or `str` for display or further calculations.
    - **Example of conversion to int**: Using the `int()` function truncates the decimal part and converts the float to an integer.
    - **Example of conversion to str**: Using the `str()` function converts the float to a string representation.

While floats do not have methods in the same way strings do, understanding and utilizing these operations and functions is crucial for numerical computations, especially when working with real numbers. The `math` module, in particular, provides a comprehensive suite of mathematical functions that extend the capabilities of float operations for more complex calculations.

Basic I/O

Basic I/O includes the reading of data from the keyboard and displaying it to the user on the screen. Displaying is done by print() and reading from the keyboard by input().

Print function

The print() function writes the value of the argument(s) it is given. [...] it handles multiple arguments, floating point-quantities, and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely:

```
print('Hello world!') # Hello world!

a = 1
print('Hello world!', a) # Hello world! 1
```

The end parameter The keyword argument end can be used to avoid the newline after the output, or end the output with a different string:

```
phrase = ['printed', 'with', 'a', 'dash', 'in', 'between']
for word in phrase:
    print(word, end='-')

# printed-with-a-dash-in-between-
```

The sep parameter The keyword sep specify how to separate the objects, if there is more than one:

```
print('cats', 'dogs', 'mice', sep=',')
# cats,dogs,mice
```

Input function

This function takes the input from the user and converts it into a string:

```
print('What is your name?')   # ask for their name
my_name = input()
print('Hi, {}'.format(my_name))
# What is your name?
# Martha
# Hi, Martha
```

input() can also set a default message without using print():

```
my_name = input('What is your name? ')  # default message
print('Hi, {}'.format(my_name))
# What is your name? Martha
# Hi, Martha
```

It is also possible to use formatted strings to avoid using .format:

```
my_name = input('What is your name? ')  # default message
print(f'Hi, {my_name}')
# What is your name? Martha
# Hi, Martha
```

Further Reading

Check out and read the help sections inside your python interactive shell

- help("ASSIGNMENT")
- help("NUMBERS")
- help("STRINGS")