Construction of Python

Identifiers

In Python, an identifier is a name that is used to identify a variable, function, class, or any other object. An identifier can consist of letters, digits, and underscores (_), but must start with a letter or an underscore. Python identifiers are case-sensitive, meaning that "my_variable" and "My_Variable" are two different identifiers.

Here are some rules to follow when naming an identifier in Python:

1. The first character of an identifier must be a letter or an underscore (_).
2. The remaining characters can be letters, digits, or underscores.
3. Identifiers can be of any length.
4. Identifiers cannot be a Python keyword. Python keywords are reserved words that have special meanings in the language, such as "if", "else", "while", and "for".
5. It's recommended to use descriptive names for identifiers that are easy to understand and convey their purpose.

Here are some examples of valid and invalid identifiers in Python:

Valid Identifiers:

```
my_variable
_my_variable
myVariable
MyVariable
my_variable_1
```

Invalid Identifiers:

```
1_variable  # Identifiers cannot start with a digit
my-variable # Identifiers cannot contain hyphens
if # if is a Python keyword
```

Constants

In Python, constants are typically defined as variables whose values are not intended to be changed during the execution of a program. By convention, constants are named in uppercase to distinguish them from other variables.

Python does not have a specific data type for constants, but they are usually represented using variables that are never modified. Here's an example:

```
PI = 3.14159
```

In this example, **"PI"** is a constant that represents the value of pi. Because the value of PI is not intended to be changed during the execution of the program, it is named in uppercase to indicate that it is a constant.

Here are some examples of valid and invalid constants in Python:

Valid constants:

```
PI = 3.14159
MAX_VALUE = 100
TAX_RATE = 0.25
```

In the examples above, **"PI", "MAX_VALUE",** and **"TAX_RATE"** are constants that are named in uppercase letters to indicate that they are not intended to be changed during the execution of the program.

Invalid constants:

```
my_variable = 10
```

In this example, **"my_variable"** is not a constant because it is a regular variable that can be modified during the execution of the program.

It's important to note that Python does not have true constants since variables can be reassigned. However, by convention, variables whose values are not intended to be changed are named in uppercase and are treated as constants.

Constants can also be defined using the **"const"** keyword, but this is not a built-in feature of Python and is not commonly used. But, there are some third-party libraries that provide a **`const` class** that can be used to define constants in Python. Here's an example:

```
from const import const

MY_CONSTANT = const(10)
```

In this example, we import the `const` class from a third-party library and use it to define a constant named **`MY_CONSTANT`** with a value of **10**.

It's worth noting that using the `const` class is not a common practice in Python, and the convention of using uppercase variable names is more widely followed to represent constants.

Keywords

keywords are reserved words that have specific meanings and functionalities in the language. These keywords cannot be used as identifiers (e.g., variable names or function names) since they serve special purposes within the language. List of Python keywords are follows

```
False       class       finally     is          return
None        continue    for         lambda      try
True        def         from        nonlocal    while
and         del         global      not         with
as          elif        if          or          yield
assert      else        import      pass
break       except      in          raise
```

Input and Output Functions

In Python, there are several functions available for input and output of data.

For Input:

- **`input()`** function is used to get input from the user. It takes an optional string as an argument, which is used to prompt the user for input. The input is returned as a string. If the user enters nothing and presses Enter, an empty string is returned.

Here's an example of using the `input()` function:

```
name = input("What is your name? ")
print("Hello, " + name + "!")
```

In this example, the user is prompted with the string "What is your name?" The user can then enter their name, and the input is assigned to the variable **`name`**. The program then uses the **`print()`** function to display a personalized greeting.

Note that the input returned by `input()` is always a string, even if the user enters a number. If you need to convert the input to a different data type, such as an integer or a float, you can use the appropriate conversion function like **`int()`** **or** **`float()`**

Here's an example:

```
age = int(input("What is your age? "))
print("Next year, you will be", age + 1)
```

In this example, the user is prompted for their age, and the input is converted to an integer using the `int()` function. The program then adds 1 to the age and displays the result using the `print()` function.

For Output:

- **`print()`** function is used to display output to the console. It takes one or more arguments and displays them on the console. Multiple arguments can be separated by commas.

Here's an example of using the `print()` function:

```
print("Hello, world!")
```

Output:

```
Hello, world!
```

You can also use special characters to format the output. For example, the % operator can be used to insert values into a string. Here's an example:

Example 1

```
age = 25
print("I am %d years old." % age)
```

Output:

```
I am 25 years old.
```

Example 2

```
name = "John"
age = 30
print("My name is %s and I am %d years old." % (name, age))
```

In this example, the **`print()`** function uses the **`%`** operator to insert the values of the `name` and `age` variables into the string. The **`%s`** specifier is used to insert a string value, while the **`%d`** specifier is used to insert an integer value.

In Python 3, it is recommended to use f-strings for formatting output:

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

Output:

```
My name is John and I am 30 years old.
```

In this example, the print() function uses an f-string to insert the values of the name and age variables into the string. The variables are enclosed in curly braces {} and prefixed with the f character to indicate that it is an f-string.

There are also other functions available for outputting data in Python, such as sys.stdout.write() to write to standard output, among others.

Data type

In Python, there are several built-in data types that can be used to represent different kinds of data. These data types include:

- Numbers: Integers, floating-point numbers, and complex numbers
- Strings: Sequences of characters
- Booleans: True or False values
- Lists: Ordered collections of objects
- Tuples: Ordered, immutable collections of objects
- Sets: Unordered collections of unique objects
- Dictionaries: Unordered collections of key-value pairs

You can use the **type()** function to determine the data type of a value or variable. Here's an example of using some of these data types:

```
# Numbers
x = 42          # integer
y = 3.14        # floating-point number
z = 2 + 3j      # complex number

print(type(x))  # <class 'int'>
print(type(y))  # <class 'float'>

# Strings
name = "Alice"
greeting = 'Hello, world!'

# Booleans
```

```
is_sunny = True
is_raining = False

# Lists
numbers = [1, 2, 3, 4, 5]
fruits = ["apple", "banana", "orange"]

# Tuples
coordinates = (10, 20)

# Sets
colors = {"red", "green", "blue"}

# Dictionaries
person = {"name": "Bob", "age": 30, "city": "New York"}
```

Python is a dynamically typed language, which means that the data type of a variable is inferred at runtime based on the value that is assigned to it. This makes it easy to work with different kinds of data, but it also means that you need to be careful when working with variables to avoid errors that can occur when types don't match.

In addition to these built-in data types, Python also supports user-defined data types, such as classes and objects, which can be used to represent more complex data structures.

Operators

In Python, operators are symbols or keywords used to perform operations on values or variables. The following are the types of operators available in Python:

- **Arithmetic operators:** used to perform arithmetic operations on numeric values.
  - `+` addition
  - `-` subtraction
  - `*` multiplication
  - `/` division
  - `%` modulus (returns the remainder of division)
  - `**` exponentiation

    - `//` floor division (returns the quotient of division rounded down to the nearest integer)
- **Assignment operators:** used to assign values to variables.
    - `=` assigns a value to a variable
    - `+=` adds a value to a variable and assigns the result
    - `-=` subtracts a value from a variable and assigns the result
    - `*=` multiplies a variable by a value and assigns the result
    - `/=` divides a variable by a value and assigns the result
    - `%=` performs modulus on a variable and assigns the result
    - `**=` performs exponentiation on a variable and assigns the result
    - `//=` performs floor division on a variable and assigns the result
- **Comparison operators:** used to compare values and return a Boolean result (`True` or `False`).
    - `==` equal to
    - `!=` not equal to
    - `<` less than
    - `>` greater than
    - `<=` less than or equal to
    - `>=` greater than or equal to
- **Logical operators:** used to combine Boolean values and return a Boolean result.
    - **`and`** returns **`True`** if both operands are **`True`**
    - **`or`** returns **`True`** if at least one operand is **`True`**
    - `not` returns the opposite Boolean value of the operand
- **Identity operators:** used to compare the memory locations of two objects and return a Boolean result.
    - `is` returns `True` if both operands refer to the same object
    - `is not` returns `True` if both operands refer to different objects
- **Membership operators:** used to test if a value is a member of a sequence and return a Boolean result.
    - `in` returns `True` if the value is in the sequence
    - `not in` returns `True` if the value is not in the sequence
- **Bitwise operators:** used to perform bitwise operations on numeric values.

- & bitwise AND
- | bitwise OR
- ^ bitwise XOR
- ~ bitwise NOT
- << bitwise left shift
- >> bitwise right shift

Here's an example of using some of these operators:

```
x = 10
y = 5

# arithmetic operators
print(x + y)  # 15
print(x - y)  # 5
print(x * y)  # 50
print(x / y)  # 2.0
print(x % y)  # 0
print(x ** y)  # 100000

# comparison operators
print(x == y)  # False
print(x != y)  # True
print(x > y)  # True
print(x <= y)  # False

# logical operators
print(x > 5 and y < 10)  # True
print(x > 5 or y > 10)  # True
print(not(x > 5))  # False

# identity operators
a = [1, 2, 3]
b = [1, 2, 3]
print(a is b) # True
```

Simple Python Program

Example 1: Print "Hello Word" Message

```
print("Hello World")
```

Output

Hello World

Example 2: Addition of two numbers

```
num1 = int(input("Enter the first number: "))
num2 = int(input("Enter the second number: "))
result = num1 + num2
print("The result is:", result)
```

Output

```
Enter the first number: 2

Enter the second number: 4

The result is: 6
```

In this Python code, the `input()` function is used to read user input, and the `int()` function is used to convert the input strings to integers. The `print()` function is used to output the result to the console.

Example 2: Swapping of two numbers

```
# take input from the user
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))

# display the values before swapping
print("Before swapping: num1 =", num1, "and num2 =", num2)

# swap the values using a temporary variable
temp = num1
num1 = num2
num2 = temp

# display the values after swapping
print("After swapping: num1 =", num1, "and num2 =", num2)
```

Output

```
Enter the first number: 3.0

Enter the second number: 4.0


Before swapping: num1 = 3.0 and num2 = 4.0
After swapping: num1 = 4.0 and num2 = 3.0
```

Example 3: Area of the circle

```
# take input from the user
radius = float(input("Enter the radius of the circle: "))

# calculate the area of the circle
area = 3.14159 * radius**2

# display the result
print("The area of the circle with radius", radius, "is", area)
```

When you run this program, it will prompt you to enter the radius of the circle. It will then use the formula `area = pi * r^2` to calculate the area of the circle (where `pi` is approximately `3.14159`). Finally, it will display the result.

For example, if you enter a radius of `5`, the program will output:

Note that we use the float() function to convert the user's input (which is initially a string) to a floating-point number. This allows us to handle decimal values as well as integers.

Example 4 : Python program to calculate the simple interest

```
# take input from the user
principal = float(input("Enter the principal amount: "))
rate = float(input("Enter the rate of interest: "))
time = float(input("Enter the time period (in years): "))

# calculate the simple interest
interest = (principal * rate * time) / 100

# display the result
print("The simple interest on a principal amount of", principal, "at a rate
of", rate, "percent per year for", time, "years is", interest)
```

When you run this program, it will prompt you to enter the principal amount, rate of interest, and time period (in years). It will then use the formula `interest = (principal * rate * time) / 100` to calculate the simple interest. Finally, it will display the result.

For example, if you enter a principal amount of `1000`, a rate of interest of `5`, and a time period of `2`, the program will output:

```
sql
The simple interest on a principal amount of 1000.0 at a rate of 5.0
percent per year for 2.0 years is 100.0
```

Note that we use the `float()` function to convert the user's input (which is initially a string) to a floating-point number. This allows us to handle decimal values as well as integers.

Expression Evaluation

Python allows you to evaluate expressions using the `eval()` function. Here's an example:

```
# evaluate an arithmetic expression
result = eval("2 + 3 * 4")
print(result)  # output: 14

# evaluate a boolean expression
result = eval("3 < 4 and 5 >= 2")
print(result)  # output: True

# evaluate a string expression
result = eval("'Hello, ' + 'World!'")
print(result)  # output: Hello, World!
```

In this example, we use the `eval()` function to evaluate three different expressions: an arithmetic expression, a boolean expression, and a string expression. The `eval()` function takes a string argument containing a valid Python expression, evaluates the expression, and returns the result.

Note that the `eval()` function can be dangerous if used improperly, since it can execute arbitrary code. You should only use `eval()` with trusted input, and never with user input or other untrusted data.

Exercise Problems

1. **Addition, subtraction, multiplication, and division of numbers**
2. **Converting units of measurement (e.g. Celsius to Fahrenheit, meters to feet)**
3. **Simple interest and compound interest calculations**
4. **Area and perimeter calculations for geometric shapes**
5. **Converting between different data types (e.g., Strings to integers, integers to strings)**
6. **Calculating simple statistics like mean, median, and mode**
7. **To find the version of Python**

8. **To display your name in the screen**
9. **To find the distance between the two points**
10. **To converts seconds into days, hours, minutes, and seconds**