Python Programming

Python Fundamentals

Python Byte Code (.pyc)

- When a python file is imported using "import" statement, it is compiled and python byte code is created (.pyc file).
- However, when a python script is executed, bytecode creation process is done implicitly. No .pyc file is created.
- To explicitly create .pyc file, one can use py_compile module.

```
python -m py_compile demo.py
```

- This command will create demo.pyc in "**pycache**" folder.
- To execute the .pyc file,

```
python pycache/demo.pyc
```

- To compile all python files into current directory, use compileall module.

```
python -m compileall
```

Identifier

- valid word which is used to perform an action
- most of the times the identifiers are lower cased
- can be
  - variable name
  - function name
  - constant
  - class name
  - keyword

- rules
  - can not start with number
    - e.g.
      - 1name is invalid identifier
      - one_name is valid identifier
  - can not contain special character like space
    - e.g.
      - first name is invalid identifier
      - first_name is valid identifier
  - may use only underscore (_)
- conventions
  - for variables: lower case
    - e.g. name, address, first_name
  - for constant: upper case
    - e.g. PI, INTEREST_RATE
  - for functions: lower case with underscore
    - e.g. is_eligible_for_voting
  - for class: lower case with first letter uppercase
    - e.g. Person, Mobile

Variable

- identifier used to store a value
- variable can not be declared explicitly
- syntax

```
<variable name> = <initial value>
```

- e.g.

```
num = 100
```

- Python doesn't have any feature for constant values.
- Python relies on convention for the same. The constant variables are declared all caps.

```
PI = 3.1415
```

Data types

- Data type is an attribute/information associated with a piece of data that tells a computer system how to interpret its value
- Understanding data types ensures that data is collected in the preferred format and the value of each property is as expected
- It helps to know what kind of operations are often performed on a variable
- In python, all data types are **inferred**
  - Data types are assigned implicitly
  - Data types will get assigned automatically (by Python itself) by looking at the CURRENT value of the variable
- Can not declare a variable with explicit data type

```
# can not declare explicit
# int num = 100   # error
```

- Literal (a.k.a. constant) is a raw data given in a variable or constant
- Python Data Types
  - **int**
    - represents the whole numbers (+ve or -ve)
    - e.g.
      - num = 100
      - myvar = -10
      - Literals: 65, 0x41, 0o101, 0b100001
  - **float**
    - represents a value with decimal

    - e.g.
        - salary = 1034.60
    - Literals: 3.14, -9.81
- **str**
    - represents a string
    - to create a string value use
        - single quotes
            - used to create single line string
            - e.g.

```
name = 'steve'
```

        - double quotes
            - used to create single line string
            - e.g.

```
last_name = "jobs"
```

        - tripe double quotes
            - used to create multi-line string
            - e.g.

```
address = """
    House no 100,
    XYZ,
    pune 411056,
    MH, India.
"""
```

- **bool**
  - represents boolean value
  - can contain one of the two values [True/False]
  - e.g.

```
can_vote = True
```

  - Literals: True, False
- **complex**
  - num = 2 + 3j
- **object**
  - stores any type.
  - Literal: None (represents Nothing)
- **collection types**
  - mylist = [11, 22, 33, 44]
  - mytuple = ('A', 'B', 'C')
  - myset = { 2.2, 4.4, 6.6 }
  - mydictionary = { 'A': 'Apple', 'B': 'Ball' }

Type Hinting

- Python is dynamic typing i.e. types are assigned dynamically.
- Hard for developers to guess type of variable in code
- Type hints increase readability of the program.

```
num: int = 123
```

- Helpful for programmers, IDEs, and type checkers to identify potential errors
- Also, helps IDE to provide intellisense.
- NOTE: Providing hint doesn't change the type. Types are still inferred by the Python at runtime.

```
num1: str = "One" # okay
num2: int = "Two" # hint ignored by Python compiler
  # type of num2 is still String
```

Type conversion

- **Implicit type conversion**
  - Automatically done by compiler if possible (avoids data loss)
  - res1 = 5 + 3.14
    - 3.14 = float
    - 5 = int --> float
    - res1 = float
  - res2 = True + 20
    - True = bool --> int
    - 20 = int
    - res2 = int
- **Explicit type conversion**
  - Programmer converts explicitly using functions like int(), float(), str(), etc.
  - Examples:

```
f = 9.81;   i = int(f)
```

```
s = "3.14";   f = float(s)
```

```
i = 103;   b = bool(i)
```

```
x = 108;   s = str(x)
```

```
s = "False";   b = bool(s) # b is True
```

- If conversion is not compatible, runtime error will raise.
  - Examples:

```
c = 2 + 3j
n = int(c)   # error
```

```
s = "One"
n = int(s)  # error
```

```
n = 108
s = "string" + n  # error: implicit conversion not done
s = "string" + str(n) # okay
```

- Conversion functions for collections as follows: list(), tuple(), set(), dict()

User Input

- input() function returns string entered by end user.
- Programmer can use explicit type casting to convert string to other type.

```
name = input("Enter Name: ")
roll = int(input("Enter Roll: "))
marks = float(input("Enter Marks: "))
```

Operators

Arithmetic

- + : addition/string concatination
- - : subtraction
- * : multiplication
- / : true division (float)
- //: floor division (int)
- **: power of

Assignment

- = : assignment
- +=, -=, *=, /= : short-hand assignment

Comparison

- == : equal to
- != : not equal
- > : greater than

- < : less than
- >=: greater than or equal to
- <=: less than or equal to

Logical

- and:
  - logical and operator
  - returns true only when both the conditions are true
  - rule
    - true and true => true
    - true and false => false
    - false and true => false
    - false and false => false

```
if (age > 20) and (age < 60):
    print(f"{age} is within the limit")
else:
    print(f"{age} is not within the limit")
```

- or:
  - logical or operator
  - returns true when one of the conditions is true
  - rule
    - true or true => true
    - true or false => true
    - false or true => true
    - false or false => false

```
if (age < 18) or (age > 70):
    print(f"{age} is too young or too old")
else:
    print(f"{age} is not yound or old")
```

- not:
    - logical not operator
    - returns complement of the condition
    - rule
        - not true => false
        - not false => true

Bitwise operators

- ~ - bitwise not
- & - bitwise and
- | - bitwise or
- ^ - bitwise xor
- << - bitwise left shift
- >> - bitwise right shift

Special operators

- is, is not - identity operators
- in, not in - membership operators

Precedence

- When multiple operators used in same expression, they are solved by their Precedence.

Operator	Description
`()`	Parentheses
`**`	Exponentiation
`+x` `-x` `~x`	Unary plus, unary minus, and bitwise NOT
`*` `/` `//` `%`	Multiplication, division, floor division, and modulus
`+` `-`	Addition and subtraction
`<<` `>>`	Bitwise left and right shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`==` `!=` `>` `>=` `<` `<=` `is` `is not` `in` `not in`	Comparisons, identity, and membership operators
`not`	Logical NOT

and	AND
or	OR

Keywords

- reserved identifiers by Python
- can not use keyword for declaring variables or functions
- e.g. if, elif, else, for, while, switch
- **pass**
  - do not do anything
  - pass the control to the next line
  - used to create empty function/class
- **def**
  - used to define a function
- **return**
  - used to return a value