Basics of Python

Assignment Statements

- Assign values to names:
- `i = 5`
- `j = 2 * i`
- `j = j + 5`
- Left-hand side: Name, Right-hand side: Expression.
- Expression operations depend on value type.

Numeric Values

- **int**: Whole numbers (e.g., 178, -3, 4283829).
- **float**: Fractional numbers (e.g., 37.82, -0.01).

Difference Between int and float

- Internally stored as a binary sequence.
- **int**: Read as a binary number.
- **float**: Stored as mantissa + exponent (scientific notation).

Operations on Numbers

- **Arithmetic**: `+`, `-`, `*`, `/` (`/` always gives a float).
- `7 / 3.5  # 2.0`
- `7 / 2    # 3.5`
- **Quotient and Remainder**: `//`, `%`
- `9 // 5  # 1`
- `9 % 5   # 4`
- **Exponentiation**: `**`
- `3 ** 4  # 81`
- **Other Operations** (from `math` module):
- `from math import *`
- `log(), sqrt(), sin()`

Names, Values, and Types

- Values have types (determining valid operations).
- Names inherit types dynamically (unlike C, C++, Java).
- Type is not fixed and can change.
- `i = 5       # int`
- `i = 7 * 1   # still int`
- `j = i / 3   # float (division creates float)`
- `i = 2 * j   # float`

- Use `type(e)` to check the type of an expression.
- Avoid assigning mixed types to the same variable.

Boolean Values

- **Values**: `True`, `False`
- **Logical Operators**: `not`, `and`, `or`
- `not True  # False`
- `x and y  # True if both x and y are True`
- `x or y  # True if at least one is True`

Comparisons

- `==`, `!=`, `<`, `>`, `<=`, `>=`
- `x == y, a != b, z < 17 * 5, n > m, i <= j + k, 19 >= 44 * d`
- Combine with logical operators:
- `n > 0 and m % n == 0`
- Assign a boolean result to a variable:
- `divisor = (m % n == 0)`

Examples

- Function to check divisibility:

```
def divides(m, n):
    if n % m == 0:
        return True
    else:
        return False
```

- Check if a number is even:

```
def even(n):
    return divides(2, n)
```

- Check if a number is odd:

```
def odd(n):
    return not divides(2, n)
```

Summary

- Values have types that determine allowed operations.
- Names inherit type dynamically from assigned values.
- Python supports `int`, `float`, and `bool` types.

Names, Values, and Types

- Values have types.
- Types determine allowed operations.
- Names inherit types from their assigned values.
- A name can hold values of different types.
- Common types: `int`, `float`, `bool`.
- Operators: `+`, `-`, `*`, `/`, `and`, `or`, `==`, `!=`, `>`, etc.

Manipulating Text

- Computation is more than just number crunching.
- Text processing is widely used in:
    - Document preparation
    - Importing/exporting spreadsheet data
    - Matching search queries to content

Strings (`str`)

- A string (`str`) is a sequence of characters.
- A single character is a string of length 1.
- No separate `char` type in Python.
- Strings can be enclosed in:
    - Single quotes `'...'`
    - Double quotes `"..."`
    - Triple quotes `'''...'''` or `"""..."""` (for multi-line strings).
    - Example:
    - `city = 'Chennai'`
    - `title = "Hitchhiker's Guide to the Galaxy"`
    - `dialogue = '''He said his favourite book is "Hitchhiker's Guide to the Galaxy"'''`

Strings as Sequences

- A string is a sequence (list) of characters.
- Characters are indexed from `0` to `n-1` (for a string of length `n`).
- Negative indices count from the end.
- `s = "hello"`
- `s[1]  # 'e'`
- `s[-2] # 'l'`

Operations on Strings

```
- **Concatenation (`+`)**: Combine two strings.
```python
s = "hello"
t = s + ", there"
print(t)  # "hello, there"
```
```

- **Length (`len()`)**: Returns the length of a string.
- `len(s)  # 5`

Extracting Substrings (Slicing)

- A **slice** extracts a part of a string.
- `s = "hello"`
- `s[1:4]  # "ell"`
- `s[i:j]` starts at `s[i]` and ends at `s[j-1]`.
- `s[:j]` starts from `s[0]` (same as `s[0:j]`).
- `s[i:]` runs until the last character (same as `s[i:len(s)]`).

Modifying Strings

- **Strings are immutable** (cannot be changed in place).
- Attempting to modify results in an error:
- `s = "hello"`
- `s[3] = "p"  # Error!`
- Instead, use slicing and concatenation:
- `s = s[0:3] + "p!"`
- `print(s)  # "help!"`

Summary

- Text values are of type `str`, a sequence of characters.
- A single character is a string of length 1.
- Characters can be accessed using their position.
- Slices extract parts of strings.
- `+` is used to concatenate strings.
- Strings cannot be modified directly (they are **immutable**).

Types of Values in Python

- **Numbers**: `int`, `float`
  - Arithmetic operations: `+`, `-`, `*`, `/`, …
- **Logical Values**: `bool` (`True`, `False`)
  - Logical operations: `not`, `and`, …
  - Comparisons: `==`, `!=`, `<`, `>`, `<=`, `>=`
- **Strings**: `str`, sequences of characters
  - Extract by position: `s[i]`, slice: `s[i:j]`
  - Concatenation: `+`, length: `len()`

Lists

- Sequences of values
- `factors = [1, 2, 5, 10]`
- `names = ["Anand", "Charles", "Muqsit"]`
- `mixed = [3, True, "Yellow"]`
- Extract values by position or slice
- `factors[3]  # 10`
- `mixed[0:2]  # [3, True]`
- Get length using `len()`
- `len(names)  # 3`

Lists vs Strings

- **Strings**: Both single positions and slices return strings
- `h = "hello"`
- `h[0] == h[0:1]  # "h"`
- **Lists**: Single position returns a value, slice returns a list
- `factors = [1, 2, 5, 10]`
- `factors[0]   # 1`
- `factors[0:1] # [1]`

Nested Lists

- Lists can contain other lists
- `nested = [[2, [37]], 4, ["hello"]]`
- `nested[0]  # [2, [37]]`
- `nested[1]  # 4`
- `nested[2][0][3]  # "l"`
- `nested[0][1:2]  # [[37]]`

Updating Lists

- **Lists are mutable**, unlike strings
- `nested = [[2, [37]], 4, ["hello"]]`
- `nested[1] = 7`
- `print(nested)  # [[2, [37]], 7, ["hello"]]`
- Changing a nested value:
- `nested[0][1][0] = 19`
- `print(nested)  # [[2, [19]], 7, ["hello"]]`

Copying Lists

- Use slicing to make a new copy
- `list1 = [1, 3, 5, 7]`
- `list2 = list1[:]`

Equality and Identity

- **Equality (`==`****)**: Checks if values are the same
- **Identity (`is`****)**: Checks if both refer to the same object
- `list1 = [1, 3, 5, 7]`
- `list2 = [1, 3, 5, 7]`
- `list3 = list2`
- `print(list1 == list2)  # True (same values)`
- `print(list2 == list3)  # True (same values)`
- `print(list2 is list3)  # True (same object)`
- `print(list1 is list2)  # False (different objects)`

Concatenation (`+`****)

- **Combining lists**
- `list1 = [1, 3, 5, 7]`
- `list2 = [4, 5, 6, 8]`
- `list3 = list1 + list2`
- `print(list3)  # [1, 3, 5, 7, 4, 5, 6, 8]`
- **Creates a new list**
- `list1 = [1, 3, 5, 7]`
- `list2 = list1`
- `list1 = list1 + [9]`
- `print(list1)  # [1, 3, 5, 7, 9]`
- `print(list2)  # [1, 3, 5, 7] (unchanged)`

Summary

- **Lists** are sequences of values.
- **Values in a list** don't have to be the same type.
- **Lists can be nested**.
- **Access elements** using indexing and slicing.
- **Lists are mutable** (can be changed in place).
- **Assignment does not copy a list**; use `[:]` to create a copy.

A Typical Python Program

- Functions are defined at the top and executed later.
- Statements execute from top to bottom.
- Computation starts with the first statement.

Messy Program Structure

- Python allows mixing function definitions and statements.
- However, this makes the code harder to read and debug.

List Manipulations in Python

1. Adding Elements

- **Append (`.append()`)** – Adds an element at the end.
- `lst = [1, 2, 3]`
- `lst.append(4)`
- `print(lst)  # [1, 2, 3, 4]`
- **Insert (`.insert()`)** – Inserts an element at a specific index.
- `lst.insert(1, 99)`
- `print(lst)  # [1, 99, 2, 3, 4]`
- **Extend (`.extend()`)** – Merges another list.
- `lst.extend([5, 6])`
- `print(lst)  # [1, 99, 2, 3, 4, 5, 6]`

2. Removing Elements

- **Remove by value (`.remove()`)** – Deletes the first occurrence of a value.
- `lst.remove(99)`
- `print(lst)  # [1, 2, 3, 4, 5, 6]`
- **Remove by index (`.pop()`)** – Deletes an element at a given index (default last).
- `lst.pop(2)`
- `print(lst)  # [1, 2, 4, 5, 6]`
- **Delete using `del`** – Removes an element or entire slice.
- `del lst[1:3]`
- `print(lst)  # [1, 5, 6]`
- **Clear (`.clear()`)** – Empties the list.
- `lst.clear()`
- `print(lst)  # []`

3. Accessing and Modifying Elements

- **Indexing** – Access an element by position.
- `lst = [10, 20, 30, 40]`
- `print(lst[2])  # 30`
- **Slicing** – Extract a portion of the list.
- `print(lst[1:3])  # [20, 30]`
- **Modifying Elements** – Change an element by index.
- `lst[2] = 99`
- `print(lst)  # [10, 20, 99, 40]`

4. Sorting and Reversing

- **Sort (`.sort()`)** – Arranges in ascending order.
- `lst = [3, 1, 4, 1, 5]`
- `lst.sort()`
- `print(lst)  # [1, 1, 3, 4, 5]`

- **Sort in Descending Order**
- `lst.sort(reverse=True)`
- `print(lst)  # [5, 4, 3, 1, 1]`
- **Reverse (`.reverse()`)** – Reverses the list order.
- `lst.reverse()`
- `print(lst)  # [5, 4, 3, 1, 1]`

5. Searching and Counting

- **Find index of an element (`.index()`)**
- `lst = [10, 20, 30, 40]`
- `print(lst.index(30))  # 2`
- **Count occurrences (`.count()`)**
- `lst = [1, 2, 3, 1, 1, 4]`
- `print(lst.count(1))  # 3`

6. Copying Lists

- **Shallow Copy (`.copy()`)**
- `new_lst = lst.copy()`
- **Deep Copy (`copy.deepcopy()`)** – Use for nested lists.
- `import copy`
- `new_lst = copy.deepcopy(lst)`

7. List Comprehensions

- **Create a new list based on a condition or transformation.**
- `squares = [x**2 for x in range(5)]`
- `print(squares)  # [0, 1, 4, 9, 16]`
- **Filtering elements**
- `evens = [x for x in range(10) if x % 2 == 0]`
- `print(evens)  # [0, 2, 4, 6, 8]`