■ Python Data Types and Data Structures

1. Python Data Types

Python is a dynamically typed language:

- You do not declare variable types (unlike Java or C).
- The type is determined by the value you assign.
- If you later assign a different kind of value, the variable's type changes.
- When this happens, Python creates a new object and the variable points to it.

```
x = 5           # int
x = 5.0         # now float (new object created)
x = "Sarah"     # now str (new object created)
x = True        # now bool (new object created)
```

Common Python data types include:

```
# Int
x = 42

# Float
z = 3.14

# String
name = "Sarah"

# Boolean
flag = True

# Tuple (immutable ordered collection)
location = (42.3601, -71.0589)

# List (mutable ordered collection)
primes = [2, 3, 5, 7, 11, 13]

# Dictionary (key-value pairs)
student = {"fname": "Sarah", "lname": "Smith"}

# Set (unique, unordered collection)
fruits = {"apples", "oranges", "pears"}
```

2. Mutable vs Immutable and References

Data types that hold a single value are sometimes called scalar or primitive types.
Data types that hold multiple values are called data structures.

Every variable in Python stores a reference to an object in memory (RAM).

Immutable types (int, float, bool, str, tuple) → Any change creates a new object (new reference).

Mutable types (list, dict, set) → Contents can be added, removed, or modified without changing the reference to the container.

Details about mutables:
- If you reassign a variable to [] , {} , or set() , you create a new object with a new reference.
- Each element inside a data structure is itself an object with its own reference.
- Replacing an element updates the reference in the container.
- Adding an element stores the new object's reference in the container.

Key takeaway:
- Immutable = object itself cannot change, any update gives you a new object.
- Mutable = the object's identity (reference) stays the same, but its contents can change.

3. Creating Data Structures

Here are 5 different ways to create tuples, sets, lists, and dictionaries.

Tuples

Tuples — 5 Ways:

```
# 1. Using parentheses
t1 = (1, 2, 3)

# 2. Without parentheses (comma separated)
t2 = 1, 2, 3

# 3. From a list
t3 = tuple([1, 2, 3])

# 4. From a string (sequence → tuple of chars)
t4 = tuple("abc")   # ('a', 'b', 'c')

# 5. Single-element tuple (must include a comma)
t5 = (42,)
```

Sets

Sets — 5 Ways:

```
# 1. Using curly braces
s1 = {1, 2, 3}

# 2. From a list
s2 = set([1, 2, 2, 3])   # {1, 2, 3}

# 3. From a string (unique characters)
s3 = set("hello")   # {'h', 'e', 'l', 'o'}

# 4. From a tuple
s4 = set((1, 2, 3, 3))   # {1, 2, 3}

# 5. Empty set (must use constructor)
s5 = set()
```

Lists

Lists — 5 Ways:

```
# 1. Using square brackets
l1 = [1, 2, 3]

# 2. From a tuple
l2 = list((1, 2, 3))

# 3. From a string
l3 = list("cat")   # ['c', 'a', 't']

# 4. Using list comprehension
l4 = [x**2 for x in range(5)]   # [0, 1, 4, 9, 16]

# 5. Empty list
l5 = []
```

Dictionaries:
Dictionaries - 5 Ways:

1. Using curly braces {}

```
d1 = {"a": 1, "b": 2, "c": 3}
print(d1)   # {'a': 1, 'b': 2, 'c': 3}
```

2. Using dict() constructor with keyword arguments

```
d2 = dict(a=1, b=2, c=3)
print(d2)   # {'a': 1, 'b': 2, 'c': 3}
```

3. Using dict() with a list of tuples

```
d3 = dict([("a", 1), ("b", 2), ("c", 3)])
print(d3)   # {'a': 1, 'b': 2, 'c': 3}
```

4. Using dictionary comprehension

```
d4 = {x: x**2 for x in range(3)}
print(d4)   # {0: 0, 1: 1, 2: 4}
```

5. Using zip() to combine two lists

```
keys = ["a", "b", "c"]
values = [1, 2, 3]
d5 = dict(zip(keys, values))
print(d5)   # {'a': 1, 'b': 2, 'c': 3}
```

Lists

List Methods

append(elem): add element to end (mutates)
```
nums = [1, 2, 3]
nums.append(4)
print(nums) # [1, 2, 3, 4]
```

extend(iterable): add elements from iterable (mutates)
```
nums = [1, 2, 3]
nums.extend([4, 5])
print(nums) # [1, 2, 3, 4, 5]
```

insert(i, elem): insert element at index (mutates)
```
nums = [1, 3, 4]
nums.insert(1, 2)
print(nums) # [1, 2, 3, 4]
```

remove(elem): remove first occurrence (mutates)
```
nums = [1, 2, 3, 2]
nums.remove(2)
print(nums) # [1, 3, 2]
```

pop([i]): remove and return element at index (mutates)
```
nums = [10, 20, 30]
x = nums.pop(1)
print(x)    # 20
print(nums) # [10, 30]
```

clear(): remove all elements (mutates)
```
nums = [1, 2, 3]
nums.clear()
print(nums) # []
```

index(elem): return index of first occurrence
```
nums = [10, 20, 30, 20]
print(nums.index(20)) # 1
```

count(elem): count occurrences
```
nums = [1, 2, 2, 3, 2]
print(nums.count(2)) # 3
```

sort(key=None, reverse=False): sort list in place (mutates)
```
nums = [3, 1, 4, 2]
nums.sort()
print(nums) # [1, 2, 3, 4]
nums.sort(reverse=True)
print(nums) # [4, 3, 2, 1]
```

reverse(): reverse list in place (mutates)
```
nums = [1, 2, 3]
nums.reverse()
```

```
print(nums) # [3, 2, 1]
```

copy(): shallow copy

```
nums = [1, 2, 3]
t = nums.copy()
print(t) # [1, 2, 3]
```

Dictionaries

Dictionary Methods

clear(): remove all items (mutates)

```
student = {"name": "Alice", "age": 20}
student.clear()
print(student)  # {}

# After clear(), nothing to loop over:
for k in student:   # (no output)
    print(k)
```

copy(): shallow copy (non-mutating)

```
student = {"name": "Alice", "age": 20}
d2 = student.copy()
print(d2)  # {'name': 'Alice', 'age': 20}

# Looping over the copy:
for k in d2:    # keys
    print(k)    # name
                # age

for v in d2.values():   # values
    print(v)            # Alice
                        # 20

for k, v in d2.items():  # key, value pairs
    print(k, v)          # name Alice
                         # age 20
```

fromkeys(iterable, value=None): build new dict (class method)

```
d = dict.fromkeys(["a", "b"], 0)
print(d)  # {'a': 0, 'b': 0}

# Looping:
for k in d:    # keys
    print(k)   # a
               # b

for k, v in d.items():   # items
    print(k, v)          # a 0
                         # b 0
```

get(key, default=None): safe lookup (non-mutating)

```
student = {"name": "Alice"}
print(student.get("name"))           # 'Alice'
print(student.get("grade", "N/A"))   # 'N/A'

# Usually used for single lookups. For multiple:
for k in student:
    print(k, student.get(k))   # name Alice
```

items(): view of (key, value) pairs (non-mutating)

```
student = {"name": "Alice", "age": 20}
print(list(student.items()))  # [('name','Alice'), ('age',20)]

# Iterating:
for k, v in student.items():
    print(f"{k} → {v}")   # name → Alice
                          # age → 20
```

keys(): view of keys (non-mutating)

```
student = {"name": "Alice", "age": 20}
print(list(student.keys()))   # ['name', 'age']

# Iterating:
for k in student.keys():
    print(k)   # name
               # age
```

pop(key[, default]): remove by key and return value (mutates)

```
student = {"name": "Alice", "age": 20}
print(student.pop("age"))         # 20
print(student)                    # {'name': 'Alice'}
print(student.pop("grade", "N/A"))  # 'N/A'
print(student)                    # {'name': 'Alice'}
# student.pop()  # ■ TypeError: missing required key

# After removal:
for k, v in student.items():
    print(k, v)   # name Alice
```

popitem(): remove & return last inserted (mutates)

```
student = {"name": "Alice", "age": 20}
print(student.popitem())  # ('age', 20)
print(student)            # {'name': 'Alice'}
# student.popitem(0)  # ■ TypeError: takes no arguments

# After popitem:
for k in student:
    print(k)   # name
```

setdefault(key, default=None): get or insert (mutates only if missing)

```
student = {"name": "Alice"}
print(student.setdefault("grade", "A"))  # 'A'
print(student)  # {'name': 'Alice', 'grade': 'A'}
print(student.setdefault("grade", "B"))  # 'A' (unchanged)
print(student)  # {'name': 'Alice', 'grade': 'A'}

# Looping after setdefault:
for k, v in student.items():
    print(k, v)   # name Alice
                  # grade A
```

update([other]): merge mapping or iterable of pairs (mutates)

```
student = {"name": "Alice"}
student.update({"age": 21, "grade": "B"})
```

```
print(student)  # {'name': 'Alice', 'age': 21, 'grade': 'B'}

# Looping:
for k in student:    # keys
    print(k)         # name
                     # age
                     # grade

for v in student.values():   # values
    print(v)                 # Alice
                             # 21
                             # B
```

values(): view of values (non-mutating)

```
student = {"name": "Alice", "age": 21}
print(list(student.values()))  # ['Alice', 21]

# Iterating:
for v in student.values():
    print(v)   # Alice
               # 21
```

Note: - list.pop([i]) allows an optional index (default is last). - dict.pop(key[, default]) requires a key (no default index).

Sets

Set Methods

add(elem): add element (mutates)

```
s = {1, 2}
s.add(3)
print(s)  # {1, 2, 3}
```

clear(): remove all elements (mutates)

```
s = {1, 2, 3}
s.clear()
print(s)  # set()
```

copy(): shallow copy (non-mutating)

```
s = {1, 2, 3}
t = s.copy()
print(t)  # {1, 2, 3}
```

difference(*others): elements in self not in others (non-mutating)

```
s = {1, 2, 3, 4}
print(s.difference({2, 3}))      # {1, 4}
```

difference_update(*others): remove elements found in others (mutates)

```
s = {1, 2, 3, 4}
s.difference_update({2, 3})
print(s)  # {1, 4}
```

discard(elem): remove elem if present; no error if absent (mutates)

```
s = {1, 2}
s.discard(2); s.discard(9)
print(s)  # {1}
```

intersection(*others): common elements (non-mutating)

```
s = {1, 2, 3}
print(s.intersection({2, 3, 4}))  # {2, 3}
```

intersection_update(*others): keep only common elements (mutates)

```
s = {1, 2, 3}
s.intersection_update({2, 3, 4})
print(s)  # {2, 3}
```

isdisjoint(other): True if no common elements (non-mutating)

```
print({1, 2}.isdisjoint({3, 4}))  # True
```

issubset(other): subset test (non-mutating)

```
print({1, 2}.issubset({1, 2, 3}))  # True
```

issuperset(other): superset test (non-mutating)

```
print({1, 2, 3}.issuperset({1, 2}))  # True
```

pop(): remove & return an arbitrary element (mutates; KeyError if empty)

```
s = {10, 20, 30}
x = s.pop()
print(x, s)  # (one of {10,20,30}, remaining set)
```

remove(elem): remove elem or KeyError if absent (mutates)

```
s = {1, 2, 3}
s.remove(2)
print(s)  # {1, 3}
# s.remove(9)  # ■ KeyError
```

symmetric_difference(other): elements in either but not both (non-mutating)

```
s = {1, 2, 3}
print(s.symmetric_difference({3, 4}))  # {1, 2, 4}
```

symmetric_difference_update(other): replace with symmetric diff (mutates)

```
s = {1, 2, 3}
s.symmetric_difference_update({3, 4})
print(s)  # {1, 2, 4}
```

union(*others): all unique elements combined (non-mutating)

```
s = {1, 2}
print(s.union({2, 3}, {3, 4}))  # {1, 2, 3, 4}
```

update(*others): add elements from others (mutates)

```
s = {1, 2}
s.update({2, 3}, {3, 4})
print(s)  # {1, 2, 3, 4}
```

Note: set.pop() removes an arbitrary element (not necessarily the largest or last). Use remove(x) to delete a specific value, or discard(x) to avoid KeyError if absent.

Tuples

Tuple Methods (tuples are immutable):

count(x): number of occurrences (non-mutating)

```
t = (1, 2, 2, 3)
print(t.count(2))  # 2
```

index(x, start=0, end=len(tuple)): first index or ValueError (non-mutating)

```
t = (1, 2, 2, 3)
print(t.index(3))     # 3
print(t.index(2, 1))  # 1
```