Numbers in Python

1. Integers

Definition: Whole numbers without a fractional or decimal part. They can be positive, negative, or zero. Examples: -5, 0, 42

2. Floats

Definition: Numbers that contain a decimal point or are expressed in scientific notation. Examples: 3.14, -0.001, 1.2e3 (which equals 1200.0)

3. Complex Numbers

Definition: Numbers in the form $a+bj$ where a is the real part, b is the imaginary part, and j represents the square root of -1 Examples: 2 + 3j, -1.5 + 0j

In [3]:
```
# Integer example
a = 10
b = -3
# Float example
x = 3.14
y = -0.5
# Complex number example
z1 = 2 + 3j
z2 = 1 - 2j
```

In [4]:
```
# 1) Addition
# Integers
print(a + b)  # 10 + (-3) = 7

# Floats
print(x + y)  # 3.14 + (-0.5) = 2.64

# Complex
print(z1 + z2)  # (2+3j) + (1-2j) = 3+1j
```

```
7
2.64
(3+1j)
```

In [5]:
```
# 2) Substraction
# Integers
print(a - b)  # 10 - (-3) = 13

# Floats
print(x - y)  # 3.14 - (-0.5) = 3.64

# Complex
print(z1 - z2)  # (2+3j) - (1-2j) = 1+5j
```

```
13
3.64
(1+5j)
```

In [6]:

```
# 3) multiplication
# Integers
print(a * b)  # 10 * (-3) = -30

# Floats
print(x * y)  # 3.14 * (-0.5) = -1.57

# Complex
print(z1 * z2)  # (2+3j) * (1-2j) = 8-1j
```

```
-30
-1.57
(8-1j)
```

In [7]:

```
# 4) Division
# Integers
print(a / b)  # 10 / (-3) = -3.333...

# Floats
print(x / y)  # 3.14 / (-0.5) = -6.28

# Complex
print(z1 / z2)  # (2+3j) / (1-2j) = -0.2+1.6j
```

```
-3.3333333333333335
-6.28
(-0.8+1.4j)
```

Operators in python

In Python, operators are symbols or keywords used to perform operations on variables and values. They can be classified into various types based on their functionality. Below is an overview of some key operators:

1. Arithmetic Operators

Used to perform mathematical operations.

In [10]:

```
#Addition
a=10
b=20
print(a+b)

#substraction
print(b-a)
```

```
30
10
```

2. Comparison Operators

Used to compare two values. These return a Boolean (True or False).

In [14]:

```
# ==    Equal to
#!=     Not equal to
# <     Less than
# >     Greater than
a=10
b=20
print(a == b)  # False
print(a != b)  # True
print(a > b)   # True
print(a < b)   # False
```

```
False
True
False
True
```

3. Logical Operators

Used to combine conditional statements.

In [16]:

```
# and: Returns True if both conditions are true (a > b) and (a > 0)
# or: Returns True if at least one condition is true
print((a > b) and (b > 0))
print((a > b) or (b < 0))
```

```
False
False
```

List in python

A list in Python is a versatile data structure used to store multiple items in a single variable. It is one of the most commonly used data types in Python and is defined by enclosing elements in square brackets ([]).

In [17]:

```
#1. Accessing Elements
#Indexing: Access elements using their position (starting from 0).
#Slicing: Access a range of elements using the syntax list[start:end:step].
my_list = [10, 20, 30, 40, 50]

# Indexing
print(my_list[0])  # 10
print(my_list[-1])  # 50 (last element)

# Slicing
print(my_list[1:4])  # [20, 30, 40]
print(my_list[:3])
```

```
10
50
[20, 30, 40]
[10, 20, 30]
```

In [20]:

```
#2. Modifying Elements
#Change an element by specifying its index.
my_list[2] = 35  # Modify the 3rd element
print(my_list)  # [10, 20, 35, 40, 50]
```

```
[10, 20, 35, 40, 50]
```

In [22]:

```
#3. Adding Elements
#append(): Adds an element to the end of the list.
```

```
#insert(): Inserts an element at a specific position.

# Append
my_list.append(60)
print(my_list)  # [10, 20, 35, 40, 50, 60]

# Insert
my_list.insert(2, 25)
print(my_list)  # [10, 20, 25, 35, 40, 50, 60]
```

```
[10, 20, 35, 40, 50, 60]
[10, 20, 25, 35, 40, 50, 60]
```

In [23]:

```
#4. Removing Elements
#remove(value): Removes the first occurrence of a value.
#pop(index): Removes an element by index (default is the last element).
#clear(): Removes all elements from the list.

# Remove
my_list.remove(25)
print(my_list)  # [10, 20, 35, 40, 50, 60]

# Pop
popped = my_list.pop(3)
print(popped)  # 40 (removed element)
print(my_list)  # [10, 20, 35, 50, 60]

# Clear
my_list.clear()
print(my_list)  # []
```

```
[10, 20, 35, 40, 50, 60]
40
[10, 20, 35, 50, 60]
[]
```

In [24]:

```
#5. Other Methods
#sort(): Sorts the list in ascending order (or descending if reverse=True is use
#reverse(): Reverses the list order.
#extend(): Adds elements from another list or iterable to the end.
#count(value): Counts occurrences of a value.
#index(value): Finds the index of the first occurrence of a value.
#copy(): Returns a shallow copy of the list

my_list = [50, 10, 40, 20, 30, 20]

# Sort
my_list.sort()
print(my_list)

# Reverse
my_list.reverse()
print(my_list)

# Extend
my_list.extend([60, 70])
print(my_list)

# Count
print(my_list.count(20))
```

```
# Index
print(my_list.index(40))

# Copy
new_list = my_list.copy()
print(new_list)
```

```
[10, 20, 20, 30, 40, 50]
[50, 40, 30, 20, 20, 10]
[50, 40, 30, 20, 20, 10, 60, 70]
2
1
[50, 40, 30, 20, 20, 10, 60, 70]
```

Tuples in python

In [26]:

```
# 1. Accessing Elements
#Indexing: Access individual elements using their position (starting from 0).
#Slicing: Extract a range of elements using the syntax tuple[start:end:step].

my_tuple = (10, 20, 30, 40, 50)

# Indexing
print(my_tuple[0])
print(my_tuple[-1])

# Slicing
print(my_tuple[1:4])
print(my_tuple[:3])
print(my_tuple[::2])
```

```
10
50
(20, 30, 40)
(10, 20, 30)
(10, 30, 50)
```

In [28]:

```
# 2. Slicing
#Slicing allows extracting specific portions of a tuple by defining the start, e
# Extract a portion
print(my_tuple[1:4])

# Reverse the tuple
print(my_tuple[::-1])
```

```
(20, 30, 40)
(50, 40, 30, 20, 10)
```

In [30]:

```
#3. Concatenation
#Tuples can be combined using the + operator, creating a new tuple.
tuple1 = (1, 2, 3, 4)
tuple2 = (5, 6, 7, 8)

# Concatenate
combined_tuple = tuple1 + tuple2
print(combined_tuple)
```

```
(1, 2, 3, 4, 5, 6, 7, 8)
```

In [32]:

```
#4. Repetition
#Tuples can be repeated using the * operator, which duplicates the elements.
repeated_tuple = tuple1 * 2
print(repeated_tuple)
```

```
(1, 2, 3, 4, 1, 2, 3, 4)
```

In [36]:

```
#5. Count
#The count() method returns the number of occurrences of a specific element.
tuple_with_duplicates = (9,8,7,6,5,4,3)

# Count occurrences of 2
print(tuple_with_duplicates.count(6))
```

```
1
```

In [40]:

```
#6. Index
#The index() method returns the first index of a specific element. If the elemen
# Find index of an element
print(tuple_with_duplicates.index(3))

# Find index of first occurrence of 2
print(tuple_with_duplicates.index(7))
```

```
6
2
```

Set in python

In Python, sets are an unordered collection of unique elements. Sets are useful for mathematical operations like union, intersection, and difference. Here's a discussion and demonstration of set operations and methods:

In [42]:

```
#1. Adding Elements
#add(element): Adds a single element to the set.
my_set = {1, 2, 3}
my_set.add(4)
print(my_set)
```

```
{1, 2, 3, 4}
```

In [43]:

```
# 2. Removing Elements
# remove(element): Removes a specific element from the set. Raises a KeyError if
# discard(element): Removes a specific element without raising an error if the e
# pop(): Removes and returns an arbitrary element from the set.
# clear(): Removes all elements from the set.

my_set = {1, 2, 3, 4}

# Remove a specific element
my_set.remove(3)
print(my_set)  # {1, 2, 4}

# Discard (doesn't raise error if element not found)
my_set.discard(5)  # No error
print(my_set)  # {1, 2, 4}

# Pop
```

```
popped = my_set.pop()
print(popped)  # 1 (arbitrary element removed)
print(my_set)  # {2, 4}

# Clear
my_set.clear()
print(my_set)  # set()
```

```
{1, 2, 4}
{1, 2, 4}
1
{2, 4}
set()
```

In [44]:

```
# 3. Union
#Combines all unique elements from two sets.
#Method: set1.union(set2)
#Operator: set1 | set2

set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(set1.union(set2))
print(set1 | set2)
```

```
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
```

In [45]:

```
# 4. Intersection
# Finds common elements between two sets.
#Method: set1.intersection(set2)
#Operator: set1 & set2

print(set1.intersection(set2))
print(set1 & set2)
```

```
{3}
{3}
```

In [47]:

```
# 5. Difference
#Finds elements in one set but not in the other.
#Method: set1.difference(set2)
#Operator: set1 - set2

print(set1.difference(set2))
print(set1 - set2)
```

```
{1, 2}
{1, 2}
```

In [46]:

```
# Create sets
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Add elements
set1.add(6)
print("After add:", set1)  # {1, 2, 3, 6}

# Remove elements
set1.remove(6)
print("After remove:", set1)  # {1, 2, 3}

# Union
```

```
print("Union:", set1.union(set2))  # {1, 2, 3, 4, 5}

# Intersection
print("Intersection:", set1.intersection(set2))  # {3}

# Difference
print("Difference:", set1.difference(set2))  # {1, 2}
```

```
After add: {1, 2, 3, 6}
After remove: {1, 2, 3}
Union: {1, 2, 3, 4, 5}
Intersection: {3}
Difference: {1, 2}
```

Dictionaries in Python

A dictionary is an unordered collection of key-value pairs. Keys must be unique and immutable (e.g., strings, numbers, tuples), while values can be of any data type.

In [53]:

```
# 1. Accessing Key-Value Pairs
#You can access the value associated with a key using square brackets [] or the

my_dict = {'a': 1, 'b': 2, 'c': 3}
print(my_dict['a'])
print(my_dict.get('b'))
print(my_dict.get('z', 'Not Found'))
```

```
1
2
Not Found
```

In [54]:

```
#2. Adding or Updating Items
#You can add new key-value pairs or update existing ones by assigning a value to

my_dict = {'a': 1, 'b': 2}
my_dict['c'] = 3  # Add a new key-value pair
my_dict['a'] = 10  # Update an existing key
print(my_dict)
```

```
{'a': 10, 'b': 2, 'c': 3}
```

In [52]:

```
#3. Removing Items
#Use pop(), popitem(), or del to remove items from a dictionary:
#pop(key) removes the item with the specified key and returns its value.
#popitem() removes and returns the last inserted key-value pair as a tuple.
#del deletes the item with the specified key.

my_dict = {'a': 1, 'b': 2, 'c': 3}
print(my_dict.pop('b'))
print(my_dict)

print(my_dict.popitem())
print(my_dict)

del my_dict['a']
print(my_dict)
```

```
2
{'a': 1, 'c': 3}
('c', 3)
{'a': 1}
{}
```

In [61]:

```
#1. keys():Returns a view object containing all the keys in the dictionary.
my_dict = {'a': 1, 'b': 2, 'c': 3}
print(my_dict.keys())

#2. get():
print(my_dict.get('b'))

#2. values():Returns a view object containing all the values in the dictionary.
print(my_dict.values())

#3. items():Returns a view object containing key-value pairs as tuples.
print(my_dict.items())

#4.pop():Removes the item with the specified key and returns its value.
print(my_dict.pop('b'))
print(my_dict)

#6. update(): Update an existing key
my_dict['d']=4
my_dict.update(my_dict)
print(my_dict)
```

```
dict_keys(['a', 'b', 'c'])
2
dict_values([1, 2, 3])
dict_items([('a', 1), ('b', 2), ('c', 3)])
2
{'a': 1, 'c': 3}
{'a': 1, 'c': 3, 'd': 4}
```

Common Python Errors and How to Fix Them

Python errors occur when the interpreter encounters something it cannot process. Below is a discussion of common Python errors, their causes, and solutions.

1. IndentationError

Cause: This error occurs when the code is not properly indented or follows inconsistent indentation. Python relies on indentation to define blocks of code.

In [62]:

```
# IndentationError: expected an indented block
def greet():
print("Hello, World!")  # No indentation

#Ensure proper indentation. By convention, use 4 spaces per indentation level.
```

```
  Cell In[62], line 3
    print("Hello, World!")  # No indentation
    ^
IndentationError: expected an indented block after function definition on line 2
```

In [63]:
```
def greet():
    print("Hello, World!")  # Proper indentation
```

2. NameError

This error occurs when you try to use a variable, function, or object that has not been defined.

In [64]:
```
print(value) #Define the variable or check for typos before using it.
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[64], line 1
----> 1 print(value)

NameError: name 'value' is not defined
```

In [65]:
```
value = 200
print(value)
```

```
200
```

3. ValueError

This error occurs when a function or operation receives a value of the correct type but with an invalid value.

In [66]:
```
number = int("abc")  #the value is valid before using it. Use exception handling
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[66], line 1
----> 1 number = int("abc")

ValueError: invalid literal for int() with base 10: 'abc'
```

In [67]:
```
number=int(123)
print(number)
```

```
123
```

4. TypeError

This error occurs when an operation or function is applied to an object of an inappropriate type.

In [70]:
```
result = "25 " + 25  #string cannot convert in int
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[70], line 1
----> 1 result = "25 " + 25

TypeError: can only concatenate str (not "int") to str
```

In [71]:

```
result = "25" + str(25)  # Correct usage
print(result)
```

```
2525
```

5. IndexError

This error occurs when trying to access an index that is out of range for a list or other indexed collection.

In [73]:

```
my_list = [1, 2, 3]
print(my_list[5])  #Check the length of the collection before accessing an index
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[73], line 2
      1 my_list = [1, 2, 3]
----> 2 print(my_list[5])

IndexError: list index out of range
```

In [74]:

```
my_list = [1, 2, 3,4,5,6,7,8]
print(my_list[5])
```

```
6
```

6. KeyError

This error occurs when trying to access a key that does not exist in a dictionary.

In [78]:

```
my_dict = {'a':1,'b':2,'c':3}
print(my_dict['e'])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[78], line 2
      1 my_dict = {'a':1,'b':2,'c':3}
----> 2 print(my_dict['e'])

KeyError: 'e'
```

In [79]:

```
my_dict = {'a':1,'b':2,'c':3,'d':4,'e':5}
print(my_dict['e'])
```

```
5
```

In [1]:

```
%history -f main1.py
```

In [ ]: