Useful DOS commands

- **dir** - list of files and folders in current location (**dir /a:hd** - list all files including hidden)
- **cd <nameoffolder>** - change location (relatively or absolutely)
- **x:** - change logical drive, where x is the drive letter
- **mkdir <foldername>** - create folder "foldername" in the current location
- **rmdir /s /q <foldername>** - delete the folder "foldername" in the current location (additional option /s – including subfolders, /q – quiet mode)
- **more <filename>** or **type <filename>** - display content of the file "filename"
- **del <filename>** - delete the file "filename"
- **copy <filename> <folderorfile>** - copy file "filename" to a folder or duplicate it
- **wmic logicaldisk get name** – get of list drives

To display commands: **help**

To display command options: **/?**

name – file name, **name\** - foldername, **.** – current folder, \ - folder separator

Useful Linux shell commands

- **ls** - list of files and folders in current location (**ls -a** - list all files including hidden)
- **cd <nameoffolder>** - change location (relatively or absolutely)
- **mkdir <foldername>** - create folder “foldername” in the current location
- **rmdir /s /q <foldername>** - delete the folder “foldername” in the current location (additional option /s – including subfolders, /q – quiet mode)
- **more <filename>** or **cat <filename>** - display content of the file “filename”
- - display content of the file “filename”
- **rm <filename>** - delete the file “filename”
- **cp <filename> <folderorfile>** - copy file “filename” or duplicate
- **/** - root folder
- **~** - user home folder

To display commands: **help**

To display command options: **--help**

name – file name, **name/** - foldername, **.** – current folder, **/** - folder separator

Data Types in Python

Python has several built-in data types. Here are some commonly used ones:

- **NoneType**: empty variable, value is `None`
- **Integers** (`int`): Integers represent whole numbers without decimals. For example: `5`, `-10`, `1000`.
- **Floating-Point Numbers** (`float`): Floating-point numbers represent decimal numbers. For example: `3.14`, `1.23`, `-0.5`.
- **Strings** (`str`): Strings represent sequences of characters enclosed in single quotes (`'  '`) or double quotes (`"  "`). For example: `'Hello'`, `"Python"`, `'123’`.
- **Booleans** (`bool`): Booleans represent either `True` or `False` values, which are used for logical operations and conditional statements.

Integer (int)

In Python, integers have arbitrary precision, meaning they can represent numbers of any size as long as there is enough memory available. There is no upper limit on the size of integers in Python, apart from the available memory on your system.

Integer operations

Arithmetic Operations:

- Addition (+), Subtraction (-), Multiplication (*), and Division (/):

```
a = 5
b = 3
sum = a + b # 8
difference = a - b # 2
product = a * b # 15
quotient = a / b # 1.6666666666666667
```

- Modulo (%) and Floor Division (//):
    - Modulo (%) returns the remainder of the division operation.
    - Floor division (//) performs division and returns the quotient, rounded down to the nearest whole number.

```
a = 10
b = 3
remainder = a % b # 1
quotient = a // b # 3
```

- Exponentiation (**): Exponentiation raises a number to a power.

```
a = 2
b = 3
result = a ** b # 8
```

Integer operations

- **Bitwise operations**:
  - AND (&)
  - OR (|)
  - XOR (^)
  - complement (~)
  - left shift (<<)
  - right shift (>>)

- **Assignment operators**:
  - =
  - += to perform arithmetic operations and update the value of an integer variable in-place.
  - -=
  - *=
  - /=

```
a = a + 10
a += 10

a = a - 10
a -= 10
```

Floating-Point Number (*float*)

Floating-point numbers in Python are implemented using the IEEE 754 double-precision format, which provides a certain precision. The limits on floating-point numbers are defined by the standard and depend on the size of the floating-point representation (typically 64 bits). The maximum and minimum values that can be represented by a float can be accessed through the `sys.float_info` object in Python:

```
import sys


# Maximum representable floating-point number
print(sys.float_info.max)
# Minimum representable floating-point number
print(sys.float_info.min)
```

Float operations

Arithmetic Operations:

- Addition (+), Subtraction (-), Multiplication (*), and Division (/):

```
a = 3.5
b = 2.1
sum = a + b # 5.6
difference = a - b # 1.4
product = a * b # 7.35
quotient = a / b # 1.6666666666666667
```

- Exponentiation (**): Exponentiation raises a number to a power.

```
a = 2.0
b = 3.3
result = a ** b # 9.849155306759329
```

- Other Operations:
  - Truncating Decimal Places: You can truncate the decimal places of a float to a specific precision using the `round()` function.

```
num = 3.14159
rounded_num = round(num, 2) # 3.14
```

String (str)

The `str` data type represents sequences of characters.

- Strings in Python can hold any combination of characters, including letters, numbers, symbols, and whitespace.
- The length of a string is limited by the available memory on your system.
- Strings represent sequences of characters enclosed in single quotes (`'  '`) or double quotes (`"  "`). Also, we can use (`'''  '''`) or double quotes (`"""  """`).
- There are raw, binary and formatted strings: `r'  '`, `b'  '`, `f'  '` (using single double quotes)

String operations

- **String Concatenation** (+): The + operator is used to concatenate or join two strings together.

```
str1 = "Hello“
str2 = "World“
result = str1 + " " + str2 # "Hello World“
```

- **String Repetition** (*): The * operator can be used to repeat a string multiple times.

```
str1 = "Hello“
result = str1 * 3 # "HelloHelloHello"
```

- **String Length** (`len()`): The `len()` function returns the length of a string, which is the number of characters it contains.

```
str1 = "Hello“
length = len(str1) # 5
```

String slicing

Python uses 0-based indexing, where the first character has an index of 0!

- **Accessing Characters by Index**: You can access individual characters of a string by their index using square brackets (`[]`).

```
str1 = "Hello“
char = str1[0] # "H“
```

- **String Slicing:** You can extract a substring from a string using slicing, which allows you to specify a range of indices.

Slicing syntax: `[start:end:step]`. The start index is inclusive, while the end index is exclusive.

  - `[start:end]` means that `step = 1`
  - `[start:]` means from `start` to the end element
  - `[:end]` means from the starting element to `end`
  - Negative value of `start` or `end` means element position from the end

```
str1 = "Hello World“
substr = str1[6:11] # "World"
```

Useful String Methods

Python provides several built-in string methods that allow you to perform various operations on strings, such as:

- `upper()`, `lower()`: Convert the string to uppercase or lowercase.
- `strip()`, `lstrip()`, `rstrip()`: Remove leading and trailing whitespace.
- `split()`: Split the string into a list of substrings based on a delimiter.
- `join()`: Join a list of strings into a single string.
- `replace()`: Replace occurrences of a substring with another substring.
- `find()`, `index()`: Find the index of a substring within the string.
- `count()`: Count the occurrences of a substring in the string.
- And many more.

```
str1 = "Hello, World!“
upper_str = str1.upper() # "HELLO, WORLD!“
stripped_str = str1.strip() # "Hello, World!“
substr_list = str1.split(",") # ["Hello", " World!"]
replaced_str = str1.replace("Hello", "Hi") # "Hi, World!"
```

Boolean (bool)

The bool data type represents logical values, which can be either:

`True`

or

`False`

Complex numbers (complex)

Use the symbol `j` to define imaginary part of a complex number

```
z = 1j+3
```

Or

```
# Creates a complex number with real part 2 and imaginary part 3
z1 = complex(2, 3)
# Creates a complex number with real part 1.5 and imaginary part -2.7
z2 = complex(1.5, -2.7)
# Creates a complex number with real part 0 and imaginary part 1
z3 = complex(0, 1)
# Creates a complex number with real part 5 and imaginary part 2 (converted from strings)
z4 = complex('5', '2’)
```

Function to deal with complex numbers

- `abs(z)`
- `arg(z)`
- `z.real`
- `z.imag`

Type conversion (casting types)

- to convert a float or a string to an integer using the `int()` function.

```
x = int(5.32) # 5
y = int("3") # 3
z = int("3.14") # Error
```

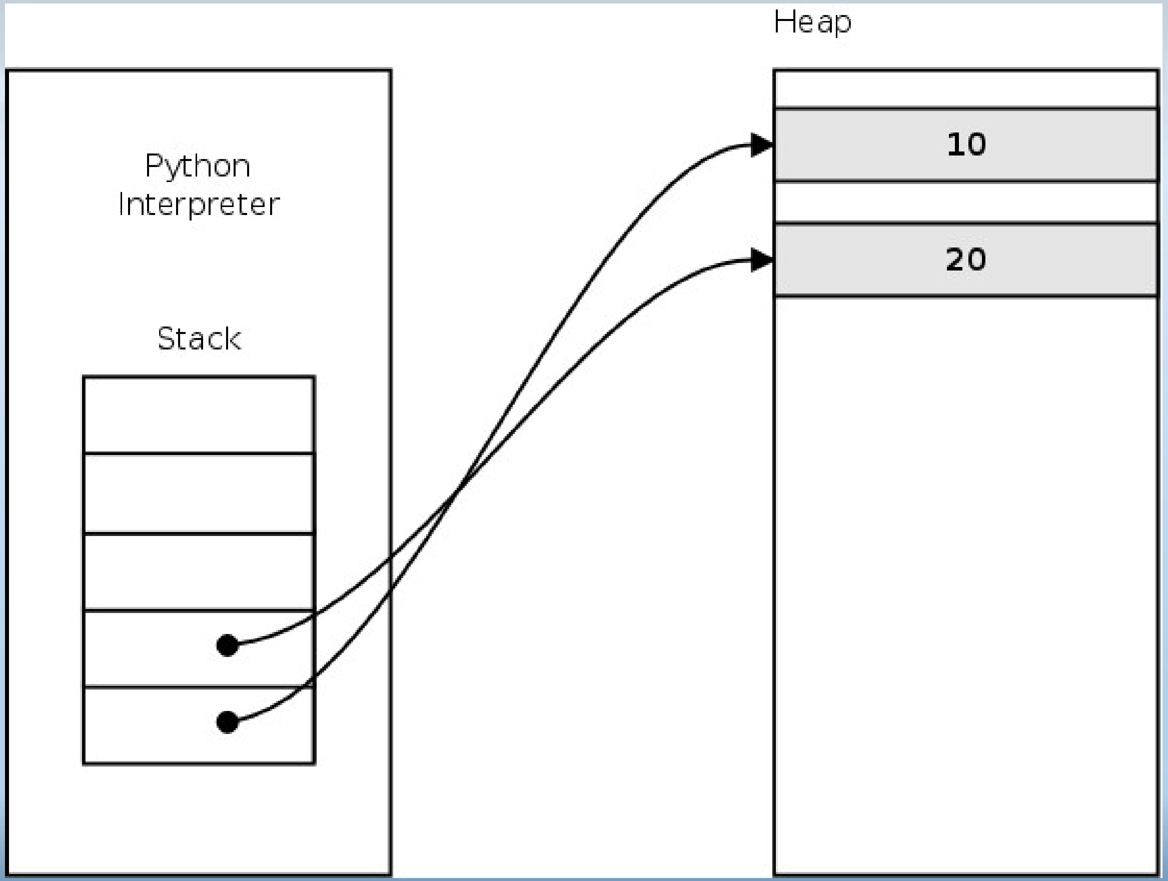
- to convert an integer or a string to a float using the `float()` function.

```
x = float(5) # 5.0
y = float("3.14") # 3.14
```

- to convert an integer or a float to a string using the `str()` function.

```
x = str(5) # "5"
y = str(3.14) # "3.14"
```

Python's Stack Machine



User input and output

- **input()** function is used for the console input.

Syntax: `variable = input(prompt)`

`prompt` is an optional parameter that specifies the message or prompt displayed to the user before accepting input. It is a string and can be omitted if no prompt is needed.

```
name = input("Enter your name: ")
age = input("Enter your age: ")
age = int(age) # Convert the input string to an integer
```

User input and output

- **print()** function is used for displaying output to the console or file. The syntax of the print() function:

```
print(format, value1, value2, ..., sep=' ', end='\n', file=sys.stdout,
flush=False)
```

The `print()` function can accept multiple arguments separated by commas. The main parameters:

- `value1, value2, ...`: These are the values or expressions that you want to display. They can be variables, strings, numbers, or other data types.
- `sep=' '` (optional) : This parameter specifies the separator between multiple values. By default, it is set to a space character. You can change it to any string you want.
- `end='\n'` (optional) : This parameter specifies the ending character(s) after the values are printed. By default, it is set to a newline character (`\n`). You can change it to any string you want.
- `file=sys.stdout` (optional) : This parameter specifies the file-like object where the output will be written. By default, it is set to the standard output (`sys.stdout`), which corresponds to the console. You can redirect the output to a file by specifying a file object.
- `flush=False` (optional) : This parameter determines whether the output should be flushed immediately. By default, it is set to False, meaning the output is buffered. If set to True, the output will be flushed immediately.

Formatted output

The % operator (placeholder replacement operator) creates formatted strings, also known as "old-style string formatting." It allows you to insert values into a string template by specifying placeholders and providing corresponding values:

- **Simple Value Insertion**: Use %s as a placeholder to insert a string value, %d for integers, %f for floats, and %r for any other value. The values to be inserted are provided after the % operator as a tuple.

```
name = "Alice"
age = 25
message = "My name is %s and I am %d years old." % (name, age)
print(message) # Output: My name is Alice and I am 25 years old.
```

- **Formatting Numeric Values**: Specify formatting options for numeric values using %d, %f, etc., along with format specifies. Format specifies allow you to control the width, precision, alignment, and other aspects of the output.

```
num = 3.14159
formatted_num = "The value of pi is approximately %.2f" % num
print(formatted_num) # Output: The value of pi is approximately 3.14
```

- **Multiple Value Insertion**: If you have multiple values to insert, provide them as a tuple after the % operator.

```
item = "apple"
quantity = 5
price = 0.99
receipt = "You bought %d %ss for a total of $%.2f." % (quantity, item, quantity * price)
print(receipt) # Output: You bought 5 apples for a total of $4.95.
```

Formatted string

Formatted strings provide a convenient and readable way to construct strings that include dynamic values. They eliminate the need for concatenation and make the code more concise. By using formatted strings, you can easily create formatted output for logging, user prompts, error messages, and other scenarios where dynamic string construction is required.

- Basic Syntax:
    - To create a formatted string, you precede the string with the letter `f` or `F`.
    - Inside the formatted string, you can include expressions or variables enclosed in curly braces `{}`.
    - The expressions or variables within the curly braces will be evaluated and their values will be inserted into the string.

```
name = "Alice“
age = 25
# Example 1
print(f"My name is {name} and I am {age} years old.")
# Example 2
result = f"The sum of 3 and 4 is {3 + 4}.“
print(result)
```

Formatted string

- Expressions and Variables:
  - Inside the curly braces, you can include any valid Python expression or variable name.
  - The expressions will be evaluated at runtime and their values will be converted to strings and inserted into the formatted string.

```
x = 5
y = 2
# Example 1
print(f"The value of x is {x}.")
# Example 2
print(f"The result of {x} + {y} is {x + y}.")
```

- Formatting Options:
  - You can specify formatting options within the curly braces to control how the values are displayed.
  - This includes specifying the width, precision, alignment, and other formatting details.
  - The formatting options follow a similar syntax as the `str.format()` method, with a colon : separating the expression/variable and the formatting specifications.

```
x = 3.14159
# Example 1
print(f"The value of x is {x:.2f}.") # Display x with 2 decimal places
# Example 2
name = "Bob“
print(f"Hello, {name:10}!") # Right-align the name within a field of width 10
```

Introduction to Python (*hour 3*)

- Lists and list operations.
- Accessing and modifying list elements.
- List methods and common list operations.

List

In Python, a list is a built-in data structure that allows you to store and organize a collection of items. Lists are mutable, meaning you can change, add, or remove elements after the list is created. Here's an overview of lists and common operations:

- **Creating** a list: You can create a list by enclosing items within square brackets `[ ]` separated by commas:

```
fruits = ['apple', 'banana', 'orange']
```

- **Creating** an empty list: You can use `[]` or `list()` :

```
fruits = []
```

- **Accessing** list elements: You can access individual elements in a list using their index. **Indexing starts at 0 for the first element**:

```
print(fruits[0]) # Output: 'apple‘
print(fruits[2]) # Output: 'orange'
```

- **Modifying** list elements: Lists are mutable, so you can modify individual elements by assigning new values to specific indices:

```
fruits[1] = 'grape‘
print(fruits) # Output: ['apple', 'grape', 'orange']
```

List methods and common operations

Python provides several built-in functions for performing common **list** operations:

- `len()`: Returns the number of elements in the list.

Python provides several built-in operations for performing common **list** operations:

- `+` (concatenation): Concatenates two lists to create a new list .
- `*` (repetition): Creates a new list by repeating the elements of an existing list .

Python provides several built-in methods for performing common **list** operations:

- `append()`: Adds an element to the end of the list.
- `extend()`: Appends all elements from another list to the end.
- `insert()`: Inserts an element at a specific position.
- `remove()`: Removes the first occurrence of a specified element.
- `pop()`: Removes and returns the element at a specified position.
- `index()`: Returns the index of the first occurrence of a specified element.
- `count()`: Returns the number of occurrences of a specified element.
- `sort()`: Sorts the list in ascending order.
- `reverse()`: Reverses the order of the list.

List methods and common operations

Here's an example demonstrating some of **list** methods:

```
fruits = ['apple', 'banana', 'orange']
fruits.append('grape')
print(fruits) # Output: ['apple', 'banana', 'orange', 'grape']

fruits.extend(['kiwi', 'pineapple'])
print(fruits) # Output: ['apple', 'banana', 'orange', 'grape', 'kiwi', 'pineapple']

fruits.insert(2, 'mango')
print(fruits) # Output: ['apple', 'banana', 'mango', 'orange', 'grape', 'kiwi', 'pineapple']

fruits.remove('banana')
print(fruits) # Output: ['apple', 'mango', 'orange', 'grape', 'kiwi', 'pineapple']

fruits.pop(0) # returns 'apple'
print(fruits) # Output: ['mango', 'orange', 'grape', 'kiwi', 'pineapple']

print(fruits.index('kiwi')) # Output: 2
print(fruits.count('orange')) # Output: 1
fruits.sort()
fruits.reverse()
print(fruits) # Output: ['pineapple', 'mango', 'kiwi', 'grape']
```

Tuple

Tuples are another built-in data structure in Python that are similar to lists but with one key difference: tuples are immutable, meaning their elements cannot be changed once the tuple is created:

- **Creating** a tuple: You can create a list by enclosing items within square brackets `( )` separated by commas:

```
fruits = ('apple', 'banana', 'orange‘)
```

- **Creating** an empty tuple: You can use `()`:

```
fruits = ()
```

- **Accessing** tuple elements: You can access individual elements in a list using their index. **Indexing starts at 0 for the first element**:

```
print(fruits[0]) # Output: 'apple'
print(fruits[2]) # Output: 'orange'
```

- **Modifying** tuple elements: **tuples are immutable, so you cannot modify individual elements**:

```
fruits[1] = 'grape‘# This will raise an error
```

Tuple methods and common operations

Python provides several built-in functions for performing common **tuple** operations:
- `len()`: Returns the number of elements in the tuple.

Python provides several built-in operations for performing common **tuple** operations:
- `+` (concatenation): Concatenates two tuples to create a new tuple.
- `*` (repetition): Creates a new tuple by repeating the elements of an existing tuple.

Python provides several built-in methods for performing common **tuple** operations:
- `count()`: Returns the number of occurrences of a specified element.
- `index()`: Returns the index of the first occurrence of a specified element.

List methods and common operations

Here's an example demonstrating some of **tuple** methods:

```
fruits = ('apple', 'banana', 'orange‘)

print(fruits.index('kiwi')) # Output: 2
print(fruits.count('orange')) # Output: 1

print(fruits) # Output: ('pineapple', 'mango', 'kiwi', 'grape')

another_tuple = ('mango', 'kiwi')
combined_tuple = fruits + another_tuple
print(combined_tuple) # Output: ('apple', 'banana', 'orange', 'mango', 'kiwi')

repeated_tuple = another_tuple * 3
print(repeated_tuple) # Output: ('mango', 'kiwi', 'mango', 'kiwi', 'mango', 'kiwi')
```

Dictionary

Dictionaries are another built-in data structure in Python that allow you to store and retrieve key-value pairs. Dictionaries are mutable and unordered, meaning the elements are not stored in a specific order and can be modified. Here's an overview of dictionaries and common operations:

- **Creating** a dictionary: You can create a list by enclosing items within square brackets `{ }` separated by commas:

```
student = {'name': 'John', 'age': 20, 'grade': 'A'}
```

- **Creating** an empty dictionary: You can use `{}` or `dict()` :

```
student = {}
```

- **Accessing** dictionary elements: You can access individual elements in a dictionary using the corresponding key:

```
print(student['name'])  # Output: 'John'
print(student['age'])   # Output: 20
```

- **Modifying** dictionary elements: dictionary are mutable, so you can modify the value of a specific key by assigning a new value to it:

```
student['age'] = 21
print(student)  # Output: {'name': 'John', 'age': 21, 'grade': 'A'}
```

Dictionary methods and common operations

Python provides several built-in functions for performing common **dictionary** operations:

- `len()`: Returns the number of elements in the dictionary.

Python provides several built-in methods for performing common **dictionary** operations:

- `keys()`: Returns a list of all keys in the dictionary.
- `values()`: Returns a list of all values in the dictionary.
- `items()`: Returns a list of key-value tuples in the dictionary.
- `get()`: Returns the value associated with a specific key. If the key is not found, it returns a default value.
- `pop()`: Removes and returns the value associated with a specific key.
- `update()`: Updates the dictionary with the key-value pairs from another dictionary.

Dictionary methods and common operations

Here's an example demonstrating some of **dictionary** methods:

```
student = {'name': 'John', 'age': 20, 'grade': 'A'}
print(student.keys())   # Output: ['name', 'age', 'grade']

print(student.values()) # Output: ['John', 20, 'A']

print(student.items())  # Output: [('name', 'John'), ('age', 20), ('grade', 'A')]

print(student.get('age'))          # Output: 20

print(student.get('address'))      # Output: None

print(student.get('address', ''))  # Output: ''student.pop('grade')

print(student)  # Output: {'name': 'John', 'age': 20}

student.update({'grade': 'B', 'address': '123 Main St'})
print(student)  # Output: {'name': 'John', 'age': 20, 'grade': 'B', 'address': '123 Main St'}
```

Set

Sets are another built-in data structure in Python that represent an unordered collection of unique elements. Sets are mutable, meaning you can add or remove elements, but they do not support indexing or storing duplicate values. Here's an overview of sets and common operations:

- **Creating** a set: You can create a list by enclosing items within square brackets `{ }` separated by commas:

```
fruits = {'apple', 'banana', 'orange'}
```

- **Creating** an empty set: You can use `{}` or `set()` :

```
fruits = {}
```

- **Accessing** set elements: **You cannot access individual elements in a set!**

- **Modifying** set elements: set are mutable, so you can modify the set value:

```
fruits.add('mango')
fruits.update(['kiwi', 'pineapple'])
fruits.remove('banana')
fruits.discard('grape')
```

Set methods (examples)

Python provides several built-in functions for performing common **set** operations:
- `len()`: Returns the number of elements in the dictionary.

Python provides several built-in methods for performing common **set** operations:
- `union()`: Returns a new set containing all unique elements from both sets.
- `intersection()`: Returns a new set containing common elements between two sets.
- `difference()`: Returns a new set containing elements present in the first set but not in the second set.
- `symmetric_difference()`: Returns a new set containing elements present in either of the sets but not in both.
- `issubset()`: Checks if a set is a subset of another set.
- `issuperset()`: Checks if a set is a superset of another set.

Set methods

Here's an example demonstrating some of **set** methods:

```
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}

union_set = set1.union(set2)
intersection_set = set1.intersection(set2)
difference_set = set1.difference(set2)
symmetric_difference_set = set1.symmetric_difference(set2)

print(union_set)                   # Output: {1, 2, 3, 4, 5, 6, 7, 8}
print(intersection_set)            # Output: {4, 5}
print(difference_set)              # Output: {1, 2, 3}
print(symmetric_difference_set)    # Output: {1, 2, 3, 6, 7, 8}

print(set1.issubset(set2))         # Output: False
print(set1.issuperset(set2))       # Output: False

print(len(set1))                   # Output: 5
```

Set operations I

Python provides several built-in operations for performing common **set** operations:

- Union (`|` or `union()`): Combines two sets to create a new set that contains all unique elements from both sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1 | set2
# or union_set = set1.union(set2)
print(union_set)  # Output: {1, 2, 3, 4, 5}
```

- Intersection (`&` or `intersection()`): Creates a new set that contains only the elements that are common to both sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
intersection_set = set1 & set2
# or intersection_set = set1.intersection(set2)
print(intersection_set)  # Output: {3}
```

- Difference (`-` or `difference()`): Creates a new set that contains the elements present in the first set but not in the second set.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
difference_set = set1 - set2
# or difference_set = set1.difference(set2)
print(difference_set)  # Output: {1, 2}
```

Set operations II

- Symmetric Difference (`^` or `symmetric_difference()`): Creates a new set that contains the elements present in either of the sets, but not in both sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
symmetric_difference_set = set1 ^ set2
# or symmetric_difference_set = set1.symmetric_difference(set2)
print(symmetric_difference_set)  # Output: {1, 2, 4, 5}
```

- Subset (`<=` or `issubset()`): Checks if one set is a subset of another set (contains all the elements of the other set).

```
set1 = {1, 2}
set2 = {1, 2, 3, 4, 5}
print(set1 <= set2)  # Output: True
# or print(set1.issubset(set2))
```

- Superset (`>=` or `issuperset()`): Checks if one set is a superset of another set (contains all the elements of the other set).

```
set1 = {1, 2}
set2 = {1, 2, 3, 4, 5}
print(set1 >= set2) # Output: True
# or print(set1.issuperset(set2))
```