Python Review

What is Python?

- **Introduction**: Python is a general-purpose programming language known for its simplicity and ease of use. Python is used in many fields like data science and machine learning, web development, scripting and automation, embedded systems and IoT, and much more.
- **Common Use Cases**: Python is used in data science, machine learning, web development, cybersecurity, automation and microcomputers like the Raspberry Pi and MicroPython-compatible boards.

Python in Your Local Environment

- **Installation**: The best way to install Python on Windows, Mac, and Linux is to download the installer from the official Python website (`https://www.python.org/`).

Variables

- **Declaring Variables**: To declare a variable, you start with the variable name followed by the assignment operator (`=`) and then the data type. This can be a number, string, boolean, etc. Here are some examples:

```
name = 'John Doe'
age = 25
```

- **Naming Conventions for Variables**: Here are the naming conventions you should use for variables:
  - Variable names can only start with a letter or an underscore (_), not a number.
  - Variable names can only contain alphanumeric characters (a-z, A-Z, 0-9) and underscores (_).
  - Variable names are case-sensitive — `age`, `Age`, and `AGE` are all considered unique.
  - Variable names cannot be one of Python's reserved keywords such as `if`, `class`, or `def`.
  - Variables names with multiple words are separated by underscores. Ex. `snake_case`.

Comments

- **Single Line Comments**: These types of comments should be used for short notes you wish to leave in your code.

```
# This is a single line comment
```

- **Multi-line Strings**: These types of strings can be used to leave larger notes or to comment out sections of code.

```
"""
This is is a multi-line string.
Here is some code commented out.

name = 'John Doe'
age = 25
"""
```

- **`print()` Function**: To print data to the console, you can use the `print()` function like this:

```
print('Hello world!') # Hello world!
```

Common Data Types in Python

- **Introduction**: Python is a dynamically-typed language like JavaScript, meaning you don't need to explicitly declare types for variables. The language knows what the type of a variable is based on what you assign to the variable.
- **Integer**: A whole number without decimals:

```
my_integer_var = 10
print('Integer:', my_integer_var) # Integer: 10
```

- **Float**: A number with decimals:

```
my_float_var = 4.50
print('Float:', my_float_var) # Float: 4.50
```

- **Complex**: A number with a real and imaginary part:

```
my_complex_var = 3 + 4j
print('Complex:', my_complex_var) # Complex: (3+4j)
```

- **String**: A sequence of characters wrapped in quotes:

```
my_string_var = 'hello'
print('String:', my_string_var) # String: hello
```

- **Boolean**: A value representing either `True` or `False`:

```
my_boolean_var = True
print('Boolean:', my_boolean_var) # Boolean: True
```

- **Set**: An unordered collection of unique elements:

```
my_set_var = {7, 5, 8}
print('Set:', my_set_var) # Set: {7, 5, 8}
```

- **Dictionary**: A collection of key-value pairs, enclosed in curly braces:

```
my_dictionary_var = {"name": "Alice", "age": 25}
print('Dictionary:', my_dictionary_var) # Dictionary: {'name': 'Alice', 'age': 25}
```

- **Tuple**: An immutable ordered collection, enclosed in parentheses:

```
my_tuple_var = (7, 5, 8)
print('Tuple:', my_tuple_var) # Tuple: (7, 5, 8)
```

- **Range**: A sequence of numbers, often used in loops:

```
my_range_var = range(5)
print(my_range_var) # range(0, 5)
```

- **List**: An ordered collection of elements that supports different data types:

```
my_list = [22, 'Hello world', 3.14, True]
print(my_list) # [22, 'Hello world', 3.14, True]
```

- **None**: A special value that represents the absence of a value:

```
my_none_var = None
print('None:', my_none_var) # None: None
```

Immutable and Mutable Types

- **Immutable Types**: These types cannot change once declared, although you can point their variables at something new, which is called reassignment. They include integer, float, complex, boolean, string, tuple, range, and `None`.
- **Mutable Types**: These types can change once declared. You can add, remove, or update their items. They include collection types such as list, set, and dictionary.
- `type()` **Function**: To see the type for a variable, you can use the `type()` function like this:

```
greeting = 'Hello there!'
age = 21

print(type(greeting)) # <class 'str'>
print(type(age)) # <class 'int'>
```

- `isinstance()` **Function**: This is used to check if a variable matches a specific data type:

```
print(isinstance('Hello world', str)) # True
print(isinstance('John Doe', int)) # False
```

Working with Strings

- **Definition**: As you recall from JavaScript, strings are immutable which means you can not change them after they have been created. In Python, you can use either single or double quotes. It is recommended to chose a rule and stick with it:

```
developer = 'Jessica'
city = 'Los Angeles'
```

- **Accessing Characters from Strings**: You can access characters from strings by using bracket notation like this:

```
my_str = "Hello world"

print(my_str[0])  # H
print(my_str[6])  # w

print(my_str[-1])  # d
print(my_str[-2]) # l
```

- **Escaping Strings**: You can use a backslash (`\`) if your string contains quotes like this:

```
msg = 'It\'s a sunny day'
quote = "She said, \"Hello!\""
```

- **String Concatenation**: To concatenate strings, you can use the `+` operator like this:

```
developer = 'Jessica'
print('My name is ' + developer + '.') # My name is Jessica
```

Another way to concatenate strings is by using the `+=` operator. This is used to perform concatenation and assignment in the same step like this:

```
greeting = 'My name is '
developer = 'Jessica.'

greeting += developer
print(greeting) # My name is Jessica.
```

- `f-strings`: This is short for formatted string literals. It allows you to handle interpolation and also do some concatenation with a compact and readable syntax:

```
developer = 'Jessica'
greeting = f'My name is {developer}.'
print(greeting) # My name is Jessica.
```

- **String Slicing**: This is when you can extract portions of a string. Here is the basic syntax:

```
str[start:stop:step]
```

The start position represents the index where the extraction should be begin. The stop position is where the slice should end. This position is non inclusive. The step position represents the interval to increment for the slicing. Here are some examples:

```
message = 'Python is fun!'

print(message[0:6])  # Python
print(message[7:])  # is fun!
print(message[::2])  # Pto sfn
```

- **Getting the Length of a String**: The `len()` function is used to return the number of the characters in the string:

```
developer = 'Jessica'

print(len(developer)) # 7
```

Working with the `in` operator

- `in` **Operator**: This returns a boolean that specifies whether the character or characters exist in the string or not:

```
my_str = 'Hello world'

print('Hello' in my_str)  # True
print('hey' in my_str)    # False
print('hi' in my_str)    # False
print('e' in my_str)  # True
print('f' in my_str)  # False
```

Common String Methods

- `str.upper()`: This returns a new string with all characters converted to uppercase:

```
developer = 'Jessica'

print(developer.upper()) # JESSICA
```

- `str.lower()`: This returns a new string with all characters converted to lowercase:

```
developer = 'Jessica'

print(developer.lower()) # jessica
```

- `str.strip()`: This returns a copy of the string with specified leading and trailing characters removed (if no argument is passed to the method, it removes leading and trailing whitespace).

```
greeting = '  hello world  '

trimmed_my_str = greeting.strip()
print(trimmed_my_str)  # 'hello world'
```

- `replace()`: This returns a new string with all occurrences of the old string replaced by a new one.

```
greeting = 'hello world'

replaced_my_str = greeting.replace('hello', 'hi')
print(replaced_my_str)  # 'hi world'
```

- `split()`: This is used to split a string into a list using a specified separator. A separator is a string specifying where the split should happen.

```
dashed_name = 'example-dashed-name'

split_words = dashed_name.split('-')
print(split_words)  # ['example', 'dashed', 'name']
```

- `join()`: This is used to join elements of an iterable into a string with a separator. An iterable is a collection of elements that can be looped over like a list, string or a tuple.

```
example_list = ['example', 'dashed', 'name']

joined_str = ' '.join(example_list)
print(joined_str)  # example dashed name
```

- `str.startswith(prefix)`: This returns a boolean indicating if a string starts with the specified prefix:

```
developer = 'Naomi'

result = developer.startswith('N')
print(result)  # True
```

- `str.endswith(suffix)`: This returns a boolean indicating if a string ends with the specified suffix:

```
developer = 'Naomi'
```

```
result = developer.endswith('N')
print(result)  # False
```

- `str.find()`: This returns the index for the first occurrence of a substring. If one is not found, then `-1` is returned:

```
developer = 'Naomi'

result = developer.find('N')
print(result)  # 0

city = 'Los Angeles'
print(city.find('New')) # -1
```

- `str.count(substring)`: This counts how many times a substring appears in a string:

```
city = 'Los Angeles'
print(city.count('e')) # 2
```

- `str.capitalize()`: This returns a new string with the first character capitalized and the other characters lowercased:

```
dessert = 'chocolate cake'
print(dessert.capitalize()) # Chocolate cake
```

- `str.isupper()`: This returns `True` if all letters in the string are uppercase and `False` if otherwise:

```
dessert = 'chocolate cake'
print(dessert.isupper()) # False
```

- `str.islower()`: This returns `True` if all letters in the string are lowercase and `False` if otherwise:

```
dessert = 'chocolate cake'
print(dessert.islower()) # True
```

- `str.title()`: This returns a new string with the first letter of each word capitalized:

```
city = 'los angeles'
print(city.title()) # Los Angeles
```

- `str.maketrans()`: This method is used to create a table of 1 to 1 character mappings for translation. It is often used with the `translate()` method which applies that table to a string and return the translated result.

```
trans_table = str.maketrans('abc', '123')
print(trans_table) # {97: 49, 98: 50, 99: 51}

result = 'abcabc'.translate(trans_table)
print(result)  # 123123
```

Common Operations used with Integers and Floats

- **Basic Math Operations**: In Python, you can do basic math operations with integers and floats including addition, subtraction, multiplication and division:

```
int_1 = 56
int_2 = 12
float_1 = 5.4
float_2 = 12.0

# Addition

print('Integer Addition:', int_1 + int_2) # Integer Addition: 68
print('Float Addition:', float_1 + float_2) # Float Addition: 17.4

# Subtraction

print('Int Subtraction:', int_1 - int_2) # Int Subtraction: 44
print('Float Subtraction:',  float_2 - float_1) # Float Subtraction: 6.6

# Multiplication

print('Int Multiplication:', int_1 * int_2) # Int Multiplication: 672
print('Float Multiplication:', float_2 * float_1) # Float Multiplication: 64.80000000000001

# Division

print('Int Division:', int_1 / int_2) # Int Division: 4.666666666666667
print('Float Division:', float_2 / float_1) # Float Division: 2.222222222222222
```

When you add a float and an integer, the result will be converted to a float like this:

```
int_1 = 56
float_1 = 5.4
```

```
print(int_1 + float_1) # 61.4
```

- **Modulus Operator** (`%`): This returns the remainder when a number is divided by another number:

```
int_1 = 56
int_2 = 12

print(int_1 % int_2) # 8
```

- **Floor Division** (`//`): This operator is used to divide two numbers and round down the result to the nearest whole number:

```
int_1 = 56
int_2 = 12

print(int_1 // int_2) # 4
```

- **Exponentiation Operator** (`**`): This operator is used to raise a number to the power of another:

```
int_1 = 4
int_2 = 2

print(int_1 ** int_2) # 16
```

- `float()` **Function**: You can use this function to convert an integer to float.

```
num = 4

print(float(num)) # 4.0
```

- `int()` **Function**: You can use this function to convert an float to an integer.

```
num = 4.0

print(int(num)) # 4
```

- `round()` **Function**: This is used to round a number to the nearest whole integer:

```
num_1 = 3.4
num_2 = 7.7
```

```
print(round(num_1)) # 3
print(round(num_2)) # 8
```

- `abs()` **Function**: This is used to return the absolute value of a number:

```
num = -13

print(abs(num)) # 13
```

- `bin()` **Function**: This is used to convert an integer to its binary representation as a string:

```
num = 56

print(bin(num))  # 0b111000
```

- `oct()` **Function**: This is used to convert an integer to its octal representation as a string:

```
num = 56

print(oct(num))  # 0o70
```

- `hex()` **Function**: This is used to convert an integer to its hexadecimal representation as a string:

```
num = 56

print(hex(num))  # 0x38
```

- `pow()` **Function**: This is used to raise a number to the power of another:

```
result = pow(2, 3)
print(result)  # 8
```

Augmented Assignments

- **Definition**: Augmented assignment combines a binary operation with an assignment in one step. It takes a variable, applies an operation to it with another value, and stores the result back into the same variable.

```
# Addition assignment
my_var = 10
my_var += 5

print(my_var) # 15
```

```
# Subtraction assignment
count = 14
count -= 3

print(count) # 11

# Multiplication assignment
product = 65
product *= 7

print(product) # 455

# Division assignment
price = 100
price /= 4

print(price) # 25.0

# Floor Division assignment
total_pages = 23
total_pages //= 5

print(total_pages) # 4

# Modulus assignment
bits = 35
bits %= 2

print(bits) # 1

# Exponentiation assignment
power = 2
power **= 3

print(power) # 8
```

There are other augmented assignment operators too, like those for bitwise operators. They include `&=`, `^=`, `>>=`, and `<<=`.

Working with Functions

- **Definition**: Functions are reusable pieces of code that take inputs (arguments) and returns an output. To call a function, you need to reference the function name followed by a set of parenthesis:

```
# Defining a function

def get_sum(num_1, num_2):
    return num_1 + num_2

result = get_sum(3, 4) # function call
print(result) # 7
```

If a function does not explicitly return a value, then the default return value is `None`:

```
def greet():
    print('hello')

result = greet() # hello
print(result) # None
```

You can also supply default values to parameters like this:

```
def get_sum(num_1, num_2=2):
    return num_1 + num_2

result = get_sum(3)
print(result) # 5
```

If you call the function without the correct number of arguments, you will get a `TypeError`:

```
def calculate_sum(a, b):
    print(a + b)

calculate_sum()

# TypeError: calculate_sum() missing 2 required positional arguments: 'a' and 'b'
```

Common Built-in Functions

- **`input()` Function**: This is used to prompt the user for some input:

```
name = input('What is your name?') # User types 'Kolade' and presses Enter
print('Hello', name) # Hello Kolade
```

- `int()` **Function**: This is used to convert a number, boolean, or a numeric string into an integer:

```
print(int(3.14)) # 3
print(int('42')) # 42
print(int(True)) # 1
print(int(False)) # 0
```

Decorators

- **Definition**: Decorators are a special kind of function in Python. They are like wrappers for other functions, so they take another function as an argument.

```
def say_hello():
    name = input('What is your name? ')
    return 'Hello ' + name

def uppercase_decorator(func):
    def wrapper():
        original_func = func()
        modified_func = original_func.upper()
        return modified_func
    return wrapper

say_hello_res = uppercase_decorator(say_hello)

print(say_hello_res())
```

Scope in Python

- **Local Scope**: This is when a variable declared inside a function or class can only be accessed within that function or class.

```
def my_func():
    num = 10
    print(num)
```

- **Enclosing Scope**: This is when a function that's nested inside another function can access the variables of the function it's nested within.

```
def outer_func():
    msg = 'Hello there!'
```

```
    def inner_func():
        print(msg)
    inner_func()

print(outer_func()) # Hello there!
```

- **Global Scope**: This refers to variables that are declared outside any functions or classes which can be accessed from anywhere in the program.

```
tax = 0.70

def get_total(subtotal):
    total = subtotal + (subtotal * tax)
    return total

print(get_total(100))  # 170.0
```

- **Built-in Scope**: Reserved names in Python for predefined functions, modules, keywords, and objects.

```
print(str(45)) # '45'
print(type(3.14)) # <class 'float'>
print(isinstance(3, str)) # False
```

Comparison Operators

- **Equal (`==`)**: Checks if two values are equal:

```
print(3 == 4) # False
```

- **Not equal (`!=`)**: Checks if two values are not equal:

```
print(3 != 4) # True
```

- **Strictly greater than (`>`)**: Checks if one value is greater than another:

```
print(3 > 4) # False
```

- **Strictly less than (`<`)**: Checks if one value is less than another:

```
print(3 < 4) # True
```

- **Greater than or equal(`>=`)**: Checks if one value is greater than or equal to another:

```
print(3 >= 4) # False
```

- **Less than or equal(`<=`)**: Checks if one value is less than or equal to another:

```
print(3 <= 4) # True
```

Working with `if`, `elif` and `else` Statements

- `if` **Statements**: These are conditions used to determine if something is true or not. If the condition evaluates to `True`, then that block of code will run.

```
age = 18

if age >= 18:
    print('You are an adult') # You are an adult
```

- `elif` **Statement**: These are conditions that come after an `if` statement. If the `elif` condition evaluates to `True`, then that block of code will run.

```
age = 16

if age >= 18:
    print('You are an adult')
elif age >= 13:
    print('You are a teenager')  # You are a teenager
```

- `else` **Clause**: This will run if no other conditions evaluate to `True`.

```
age = 12

if age >= 18:
    print('You are an adult')
elif age >= 13:
    print('You are a teenager')
else:
    print('You are a child')  # You are a child
```

You can also use nested `if` statements like this:

```
is_citizen = True
age = 25

if is_citizen:
```

```
    if age >= 18:
        print('You are eligible to vote') # You are eligible to vote
else:
    print('You are not eligible to vote')
```

Truthy and Falsy Values

- **Definition**: In Python, every value has an inherent boolean value, or a built-in sense of whether it should be treated as `True` or `False` in a logical context. Many values are considered truthy, that is, they evaluate to `True` in a logical context. Others are falsy, meaning they evaluate to `False`. Here are some examples of falsy values:

```
None
False
Integer 0
Float 0.0
Empty strings ''
```

Other values like non-zero numbers, and non-empty strings are truthy.

Working with the `bool()` Function

- **Definition**: If you want to check whether a value is truthy or falsy, you can use the built-in `bool()` function. It explicitly converts a value to its boolean equivalent and returns `True` for truthy values and `False` for falsy values. Here are a few examples:

```
print(bool(False)) # False
print(bool(0))  # False
print(bool('')) # False

print(bool(True)) # True
print(bool(1)) # True
print(bool('Hello')) # True
```

Boolean Operators and Short-circuiting

- **Definition**: These are special operators that allow you to combine multiple expressions to create more complex decision-making logic in your code. There are three Boolean operators in Python: `and`, `or`, and `not`.
- **`and` Operator**: This operator takes two operands and returns the first operand if it is falsy, otherwise, it returns the second operand. Both operands must be truthy for an expression to result in a truthy value.

```
is_citizen = True
age = 25

print(is_citizen and age) # 25
```

You can also use the `and` operator in conditionals like this:

```
is_citizen = True
age = 25

if is_citizen and age >= 18:
    print('You are eligible to vote') # You are eligible to vote
else:
    print('You are not eligible to vote')
```

- `or` **Operator**: This operator returns the first operand if it is truthy, otherwise, it returns the second operand. An or expression results in a truthy value if at least one operand is truthy. Here is an example:

```
age = 19
is_employed = False

print(age or is_employed) # 19
```

Just like with the `and` operator, you can use the `or` operator in conditionals like this:

```
age = 19
is_student = True

if age < 18 or is_student:
    print('You are eligible for a student discount') # You are eligible for a student discount
else:
    print('You are not eligible for a student discount')
```

- **Short-circuiting**: The `and` and `or` operators are known as a short-circuit operators. Short-circuiting means Python checks values from left to right and stops as soon as it determines the final result.
- `not` **Operator**: This operator takes a single operand and inverts its boolean value. It converts truthy values to `False` and falsy values to `True`. Unlike the previous operators we looked at, `not` always returns `True` or `False`. Here are some examples:

```
print(not '') # True, because empty string is falsy
print(not 'Hello') # False, because non-empty string is truthy
print(not 0) # True, because 0 is falsy
print(not 1) # False, because 1 is truthy
print(not False) # True, because False is falsy
print(not True) # False, because True is truthy
```

Here is an example of the `not` operator in a conditional:

```
is_admin = False

if not is_admin:
    print('Access denied for non-administrators.') # Access denied for non-administrators.
else:
    print('Welcome, Administrator!')
```

Python Lists

- **Introduction**: In Python, the list data type is an ordered sequence of elements that can be composed of strings, numbers or even other lists. Lists are mutable and zero based indexed.

```
cities = ['Los Angeles', 'London', 'Tokyo']
```

- **Accessing Elements in a List**: To access an element from the `cities` list, you can reference its index number in the sequence:

```
cities = ['Los Angeles', 'London', 'Tokyo']
cities[0] # Los Angeles
```

- **Accessing Elements Using Negative Indexing**: To access the last element of any list, you can use `-1` as the index number:

```
cities = ['Los Angeles', 'London', 'Tokyo']
cities[-1] # Tokyo
```

- Negative indexing is used to access elements starting from the end of the list instead of the beginning at index `0`.
- **Creating Lists Using the `list()` constructor**: Lists can also be created using the `list()` constructor. The `list()` constructor is used to convert an iterable into a list:

```
developer = 'Jessica'

print(list(developer))
# Result: ['J', 'e', 's', 's', 'i', 'c', 'a']
```

- **Finding the Length of a List**: You can use the `len()` function to get the length of a list:

```
numbers = [1, 2, 3, 4, 5]
len(numbers) # 5
```

- **List Mutability**: Lists are mutable, meaning you can update any element in the list as long as you pass in a valid index number. To update lists at a particular index, you can assign a new value to that index:

```
programming_languages = ['Python', 'Java', 'C++', 'Rust']
programming_languages[0] = 'JavaScript'
```

```
print(programming_languages) # ['JavaScript', 'Java', 'C++', 'Rust']
```

- **Index Out of Range Error**: If you pass in an index (either positive or negative) that is out of bounds for the list, then you will receive an `IndexError`:

```
programming_languages = ['Python', 'Java', 'C++', 'Rust']
programming_languages[10] = 'JavaScript'

"""
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
"""
```

- **Removing Elements from a List**: Elements can be removed from a list using the `del` keyword:

```
developer = ['Jane Doe', 23, 'Python Developer']
del developer[1]
print(developer) # ['Jane Doe', 'Python Developer']
```

- **Checking if an Element Exists in a List**: The `in` keyword can be used to check if an element exists in a list:

```
programming_languages = ['Python', 'Java', 'C++', 'Rust']

'Rust' in programming_languages # True
'JavaScript' in programming_languages # False
```

- **Nesting Lists**: Lists can be nested inside other lists:

```
developer = ['Alice', 25, ['Python', 'Rust', 'C++']]
```

- To access the nested list, you will need to access it using index `2` since lists are zero-based indexed.

```
developer = ['Alice', 25, ['Python', 'Rust', 'C++']]
developer[2] # ['Python', 'Rust', 'C++']
```

- To further access the second language from that nested list, you will need to access it using index `1`:

```
developer = ['Alice', 25, ['Python', 'Rust', 'C++']]
developer[2][1] # Rust
```

- **Unpacking Values from a List:** Unpacking values from a list is a technique used to assign values from a list to new variables. Here is an example to unpack the `developer` list into new variables called `name`, `age` and `job` like this:

```
developer = ['Alice', 34, 'Rust Developer']
name, age, job = developer
```

- **Collecting Remaining Items From a List**: To collect any remaining elements from a list, you can use the asterisk (`*`) operator like this:

```
developer = ['Alice', 34, 'Rust Developer']
name, *rest = developer
```

- If the number of variables on the left side of the assignment operator doesn't match the total number of items in the list, then you will receive a `ValueError`.
- **Slicing Lists**: Slicing is the concept of accessing a portion of a list by using the slice operator `:`. To slice a list that starts at index `1` and ends at index `3`, you can use the following syntax:

```
desserts = ['Cake', 'Cookies', 'Ice Cream', 'Pie']
desserts[1:3] # ['Cookies', 'Ice Cream']
```

- **Step Intervals**: It is also possible to specify a step interval which determines how much to increment between the indices. Here is an example if you want to extract a list of just even numbers using slicing:

```
numbers = [1, 2, 3, 4, 5, 6]
numbers[1::2] # [2, 4, 6]
```

List Methods

- **append()**: Used to add an item to the end of the list. Here is an example of using the `append()` method to add the number `6` to this `numbers` list:

```
numbers = [1, 2, 3, 4, 5]
numbers.append(6)
print(numbers) # [1, 2, 3, 4, 5, 6]
```

- **Appending lists**: The `append()` method can also be used to add one list at the end of another:

```
numbers = [1, 2, 3, 4, 5]
even_numbers = [6, 8, 10]

numbers.append(even_numbers)
print(numbers) # [1, 2, 3, 4, 5, [6, 8, 10]]
```

- **extend()**: Used to add multiple items to the end of a list. Here is an example of adding the numbers `6`, `8`, and `10` to the end of the `numbers` list:

```
numbers = [1, 2, 3, 4, 5]
even_numbers = [6, 8, 10]
```

```
numbers.extend(even_numbers)
print(numbers) # [1, 2, 3, 4, 5, 6, 8, 10]
```

- **insert()**: Used to insert an item at a specific index in the list. Here is an example of using the `insert()` method:

```
numbers = [1, 2, 3, 4, 5]
numbers.insert(2, 2.5)

print(numbers) # [1, 2, 2.5, 3, 4, 5]
```

- **remove():** Used to remove an item from the list. The `remove()` method will only remove the first occurrence of an item in the list:

```
numbers = [1, 2, 3, 4, 5, 5, 5]
numbers.remove(5)

print(numbers) # [1, 2, 3, 4, 5, 5]
```

- **pop()**: Used to remove a specific item from the list and return it:

```
numbers = [1, 2, 3, 4, 5]
numbers.pop(1) # The number 2 is returned
```

- If you don't specify an element for the `pop` method, then the last element is removed.

```
numbers = [1, 2, 3, 4, 5]
numbers.pop() # The number 5 is returned
```

- **clear()**: Used to remove all items from the list:

```
numbers = [1, 2, 3, 4, 5]
numbers.clear()

print(numbers) # []
```

- **sort()**: The `sort()` method is used to sort the elements in place. Here is an example of sorting a random list of `numbers` in place:

```
numbers = [19, 2, 35, 1, 67, 41]
numbers.sort()
```

```
print(numbers) # [1, 2, 19, 35, 41, 67]
```

- **sorted()**: Used to sort the elements in a list and return a new sorted list instead of modifying the original list.
- **reverse()**: Used to reverse the order of the elements in a list:

```
numbers = [6, 5, 4, 3, 2, 1]
numbers.reverse()

print(numbers) # [1, 2, 3, 4, 5, 6]
```

- **index()**: Used to find the first index where an element can be found in a list:

```
programming_languages = ['Rust', 'Java', 'Python', 'C++']
programming_languages.index('Java') # 1
```

- If the element cannot be found using the `index()` method, then the result will be a `ValueError`.

Tuples in Python

- **Definition**: A tuple is a Python data type used to create an ordered sequence of values. Tuples can contain a mixed set of data types:

```
developer = ('Alice', 34, 'Rust Developer')
```

- Tuples are immutable, meaning the elements in the tuple cannot be changed once created. If you try to update one of the items in the tuple, you will get a `TypeError`:

```
programming_languages = ('Python', 'Java', 'C++', 'Rust')
programming_languages[0] = 'JavaScript'

"""
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: "tuple" object does not support item assignment
"""
```

- **Accessing Elements from a Tuple**: To access an element from a tuple, use bracket notation and the index number:

```
developer = ('Alice', 34, 'Rust Developer')
developer[1] # 34
```

- Negative indexing can be used to access elements starting from the end of the tuple:

```
numbers = (1, 2, 3, 4, 5)
numbers[-2] # 4
```

- If you try to pass in an index number that exceeds or equals the length of the tuple, then you will receive an `IndexError`:

```
numbers = (1, 2, 3, 4, 5)
numbers[7]

"""
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
"""
```

- A tuple can also be created using the `tuple()` constructor. Within the constructor, you can pass in different iterables like strings, lists and even other tuples.

```
developer = 'Jessica'

print(tuple(developer))
# Result: ('J', 'e', 's', 's', 'i', 'c', 'a')
```

- **Verifying Items in a Tuple**: To check if an item is in a tuple, you can use the `in` keyword like this:

```
programming_languages = ('Python', 'Java', 'C++', 'Rust')

'Rust' in programming_languages # True
'JavaScript' in programming_languages # False
```

- **Unpacking Tuples**: Items can be unpacked from a tuple like this:

```
developer = ('Alice', 34, 'Rust Developer')
name, age, job = developer
```

- If you need to collect any remaining elements from a tuple, you can use the asterisk (`*`) operator like this:

```
developer = ('Alice', 34, 'Rust Developer')
name, *rest = developer
```

- **Slicing Tuples**: Slicing can be used to extract a portion of a tuple. For example, the items `pie` and `cookies` can be sliced into a separate tuple:

```
desserts = ('cake', 'pie', 'cookies', 'ice cream')
desserts[1:3] # ('pie', 'cookies')
```

- **Removing Items from Tuples**: Removing an item from a tuple will raise a `TypeError` as tuples are immutable:

```
developer = ('Jane Doe', 23, 'Python Developer')
del developer[1]

"""
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: "tuple" object doesn't support item deletion
"""
```

- **When to use a Tuple vs a List?**: If you need a dynamic collection of elements where you can add, remove and update elements, then you should use a list. If you know that you are working with a fixed and immutable collection of data, then you should use a tuple.

Common Tuple Methods

- `count()`: Used to determine how many times an item appears in a tuple. For example, you can check how many times the language `'Rust'` appears in the tuple:

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust')
programming_languages.count('Rust') # 2
```

- If the specified item in the `count()` function is not present at all in the tuple, then the return value will be `0`:

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust')
programming_languages.count('JavaScript') # 0
```

- If no arguments are passed to the `count()` function, then Python will return a `TypeError`.
- **index()**: Used to find the index where a particular item is present in the tuple. Here is an example of using the `index()` method to find the index for the language `'Java'`:

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust')
programming_languages.index('Java') # 1
```

- If the specified item cannot be found, then Python will return a `ValueError`.
- You can pass an optional start index to the `index()` method to specify where to start searching for the item in the tuple:

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust', 'Python')
programming_languages.index('Python', 3) # 5
```

- You can also pass in an optional end index to the `index()` method to specify where to stop searching for the item in the tuple:

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust', 'Python', 'JavaScript', 'Python')
programming_languages.index('Python', 2, 5) # 2
```

- `sorted()`: Used to sort the elements in any iterable and return a new sorted list. Here is an example of creating a new list of numbers using the `sorted()` function:

```
numbers = (13, 2, 78, 3, 45, 67, 18, 7)
sorted(numbers) # [2, 3, 7, 13, 18, 45, 67, 78]
```

- **Modifying Sorting Behavior**: You can customize the sorting behavior for an iterable using the optional `reverse` and `key` arguments. Here is an example of using the `key` argument to sort items in a tuple by length:

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust', 'Python')
sorted(programming_languages, key=len)

# Result
# ['C++', 'Rust', 'Java', 'Rust', 'Python', 'Python']
```

- You can create a new list of values in reverse order, using the `reverse` argument like this:

```
programming_languages = ('Rust', 'Java', 'Python', 'C++', 'Rust', 'Python')

print(sorted(programming_languages, reverse=True))

# Result
# ['Rust', 'Rust', 'Python', 'Python', 'Java', 'C++']
```

Loops in Python

- **Definition**: Loops are used to repeat a block of code for a set number of times.
- `for` **loop**: Used to iterate over a sequence (like a list, tuple or string) and execute a block of code for each item in that sequence. Here is an example of using a `for` loop to iterate through a list and print each language to the console:

```
programming_languages = ['Rust', 'Java', 'Python', 'C++']

for language in programming_languages:
    print(language)

"""
Result

Rust
```

```
Java
Python
C++
"""
```

- Here is an example of using a `for` loop to loop through the string `code` and print out each character:

```
for char in 'code':
    print(char)

"""
Result

c
o
d
e
"""
```

- `for` loops can be nested. Here is an example of using a nested `for` loop:

```
categories = ['Fruit', 'Vegetable']
foods = ['Apple', 'Carrot', 'Banana']

for category in categories:
    for food in foods:
        print(category, food)

"""
Result

Fruit Apple
Fruit Carrot
Fruit Banana
Vegetable Apple
Vegetable Carrot
Vegetable Banana
"""
```

- **`while` loop**: Repeats a block of code until the condition is `False`. Here is an example of using a `while` loop for a guessing game:

```
secret_number = 3
guess = 0

while guess != secret_number:
    guess = int(input('Guess the number (1-5): '))
    if guess != secret_number:
        print('Wrong! Try again.')

print('You got it!')

"""
Result

Guess the number (1-5): 2
Wrong! Try again.
Guess the number (1-5): 1
Wrong! Try again.
Guess the number (1-5): 3
You got it!
"""
```

- **`break` and `continue` statements**: Used in loops to modify the execution of a loop.
- The `break` statement is used to exit the loop immediately when a certain condition is met. Here is an example of using the `break` statement for a list of `developer_names`:

```
developer_names = ['Jess', 'Naomi', 'Tom']

for developer in developer_names:
    if developer == 'Naomi':
        break
    print(developer)
```

- The `continue` statement is used to skip that current iteration and move onto the next iteration of the loop. Here is an example to use the `continue` statement instead of a `break` statement:

```
developer_names = ['Jess', 'Naomi', 'Tom']

for developer in developer_names:
    if developer == 'Naomi':
```

```
        continue
    print(developer)
```

- Both `for` and `while` loops can be combined with an `else` clause, which is executed only when the loop was not terminated by a `break`:

```
words = ['sky', 'apple', 'rhythm', 'fly', 'orange']

for word in words:
    for letter in word:
        if letter.lower() in 'aeiou':
            print(f"'{word}' contains the vowel '{letter}'")
            break
    else:
        print(f"'{word}' has no vowels")
```

Ranges and Their Use in Loops

- **The `range()` function**: Used to generate a sequence of integers.

```
range(start, stop, step)
```

- The required `stop` argument is an integer(non-inclusive) that represents the end point for the sequence of numbers being generated. Here is an example of using the `range()` function:

```
for num in range(3):
    print(num)
```

- If a `start` argument is not specified, then the default will be `0`. By default the sequence of integers will increment by `1`. You can use the optional `step` argument to change the default increment value. Here is an example of generating a sequence of even integers from 2 up to but not including 11 (i.e., includes 10)

```
for num in range(2, 11, 2):
    print(num)
```

- If you don't provide any arguments to the `range()` function, then you will get a `TypeError`.
- The `range()` function only accepts integers for arguments and not floats. Using floats will also result in a `TypeError`:

```
ERROR!
Traceback (most recent call last):
  File "<main.py>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
```

- You can use a negative integer for the `step` argument to generate a sequence of integers in decrementing order:

```
for num in range(40, 0, -10):
    print(num)
```

- The `range()` function can also be used to create a list of integers by using it with the `list` constructor. The `list` constructor is used to convert an iterable into a list. Here is an example of generating a list of even integers between 2 and 10 inclusive:

```
numbers = list(range(2, 11, 2))
print(numbers) # [2, 4, 6, 8, 10]
```

`enumerate()` and `zip()` functions in Python

- **`enumerate()`**: used to iterate over a sequence and keep track of the index for each item in that sequence. The `enumerate()` function takes an iterable as an argument and returns an `enumerate` object that consist of the index and value of each item in the iterable.

```
languages = ['Spanish', 'English', 'Russian', 'Chinese']

for index, language in enumerate(languages):
    print(f'Index {index} and language {language}')

# Result
# Index 0 and language Spanish
# Index 1 and language English
# Index 2 and language Russian
# Index 3 and language Chinese
```

- The `enumerate()` function can also be used outside of a `for` loop:

```
languages = ['Spanish', 'English', 'Russian', 'Chinese']

print(list(enumerate(languages)))
# [(0, 'Spanish'), (1, 'English'), (2, 'Russian'), (3, 'Chinese')]
```

- The `enumerate()` function also accepts an optional `start` argument that specifies the starting value for the count. If this argument is omitted, then the count will begin at `0`.
- **`zip()`**: Used to iterate over multiple iterables in parallel. Here's an example using the `zip()` function to iterate over `developers` and `ids`:

```
developers = ['Naomi', 'Dario', 'Jessica', 'Tom']
ids = [1, 2, 3, 4]

for name, id in zip(developers, ids):
    print(f'Name: {name}')
```

```
    print(f'ID: {id}')


"""
Result

Name: Naomi
ID: 1
Name: Dario
ID: 2
Name: Jessica
ID: 3
Name: Tom
ID: 4
"""
```

List comprehensions in Python

- **Definition**: List comprehension allows you to create a new list in a single line by combining the loop and the condition directly within square brackets. This makes the code shorter and often easier to read.

```
even_numbers = [num for num in range(21) if num % 2 == 0]
print(even_numbers)
```

Iterable methods

- `filter()`: Used to filter elements from an iterable based on a condition. It returns an iterator that contains only the elements that satisfy the condition. Here is an example of creating a new list of just words longer than four characters:

```
words = ['tree', 'sky', 'mountain', 'river', 'cloud', 'sun']

def is_long_word(word):
    return len(word) > 4

long_words = list(filter(is_long_word, words))
print(long_words) # ['mountain', 'river', 'cloud']
```

- `map()`: Used to apply a function to each item in an iterable and return a new iterable with the results. Here is an example of using the `map()` function to convert a list of celsius temperatures to fahrenheit:

```
celsius = [0, 10, 20, 30, 40]
```

```
def to_fahrenheit(temp):
    return (temp * 9/5) + 32

fahrenheit = list(map(to_fahrenheit, celsius))
print(fahrenheit) # [32.0, 50.0, 68.0, 86.0, 104.0]
```

- **`sum()`**: Used to get the sum from an iterable like a list or tuple. Here is an example of using the `sum()` function:

```
numbers = [5, 10, 15, 20]
total = sum(numbers)
print(total) # Result: 50
```

- You can also pass in an optional `start` argument which sets the initial value for the summation. Here is an updated example using the `start` argument as a positional argument:

```
numbers = [5, 10, 15, 20]
total = sum(numbers, 10) # positional argument
print(total) # 60
```

- You can also choose to use the `start` argument as a keyword argument like this instead:

```
numbers = [5, 10, 15, 20]
total = sum(numbers, start=10) # keyword argument
print(total) # 60
```

Lambda functions

- **Definition**: A lambda function in Python is a concise way to create a function without a name (an anonymous function).
- Lambda functions are often used as an argument to another function. Here is an example of a lambda function:

```
numbers = [1, 2, 3, 4, 5]

even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # [2, 4]
```

- Best practices for using lambda functions include not assigning them to a variable, keeping them simple and readable, and using them for short, one-off functions.

Dictionaries

- **Dictionaries**: Dictionaries are built-in data structures that store collections of key-value pairs. Keys need to be immutable data types. This is the general syntax of a Python dictionary:

```
dictionary = {
    key1: value1,
```

```
    key2: value2
}
```

- **`dict()` Constructor**: The `dict()` constructor is an alternative way to build the dictionary. You pass a list of tuples as an argument to the `dict()` constructor. These tuples contain the key as the first element and the value as the second element.

```
pizza = dict([('name', 'Margherita Pizza'), ('price', 8.9), ('calories_per_slice', 250), ('toppings', ['mozzarella', 'basil'])])
```

- **Bracket Notation**: To access the value of a key-value pair, you can use the syntax known as bracket notation.

```
dictionary[key]
```

Common Dictionary Methods

- **`get()` Method**: The `get()` method retrieves the value associated with a key. It's similar to the bracket notation, but it lets you set a default value, preventing errors if the key doesn't exist.

```
dictionary.get(key, default)
```

- **`keys()` and `values()` Methods**: The `keys()` and `values()` methods return a view object with all the keys and values in the dictionary, respectively. A view object is a way to see the content of a dictionary without creating a separate copy of the data.

```
pizza = {
    'name': 'Margherita Pizza',
    'price': 8.9,
    'calories_per_slice': 250
}

pizza.keys()
# dict_keys(['name', 'price', 'calories_per_slice'])

pizza.values()
# dict_values(['Margherita Pizza', 8.9, 250])
```

- **`items()` Method**: The `items()` method returns a view object with all the key-value pairs in the dictionary, including both the keys and the values.

```
pizza.items()
# dict_items([('name', 'Margherita Pizza'), ('price', 8.9), ('calories_per_slice', 250)])
```

- **`clear()` Method**: The `clear()` method removes all the key-value pairs from the dictionary.

```
pizza.clear()
```

- **`pop()` Method**: The `pop()` method removes the key-value pair with the key specified as the first argument and returns its value. If the key doesn't exist, it returns the default value specified as the second argument. If the key doesn't exist and the default value is not specified, a `KeyError` is raised.

```
pizza.pop('price', 10)
pizza.pop('total_price') # KeyError
```

- **`popitem()` Method**: In Python 3.7 and above, the `popitem()` method removes the last inserted item.

```
pizza.popitem()
```

- **`update()` Method**: The `update()` method updates the key-value pairs with the key-value pairs of another dictionary. If they have keys in common, their values are overwritten. New keys will be added to the dictionary as new key-value pairs.

```
pizza.update({ 'price': 15, 'total_time': 25 })
```

Looping Over a Dictionary

- **Iterating Over Values**: If you need to iterate over the values in a dictionary, you can write a `for` loop with `values()` to get all the values of a dictionary.

```
products = {
    'Laptop': 990,
    'Smartphone': 600,
    'Tablet': 250,
    'Headphones': 70,
}

for price in products.values():
    print(price)
```

Output:

```
990
600
250
70
```

- **Iterating Over Keys**: If you need to iterate over the keys in the `products` dictionary above, you can write `products.keys()` or `products` directly.

```
for product in products.keys():
    print(product)
```

```
for product in products:
    print(product)
```

Output:

```
Laptop
Smartphone
Tablet
Headphones
```

- **Iterating Over Key-Value Pairs**: If you need to iterate over the keys and their corresponding values simultaneously, you can iterate over `products.items()`. You get individual tuples with the keys and their corresponding values.

```
for product in products.items():
    print(product)
```

Output:

```
('Laptop', 990)
('Smartphone', 600)
('Tablet', 250)
('Headphones', 70)
```

To store the key and value in separate loop variables, you need to separate them with a comma. The first variable stores the key, and the second stores the value.

```
for product, price in products.items():
    print(product, price)
```

Output:

```
Laptop 990
Smartphone 600
Tablet 250
Headphones 70
```

- **`enumerate()` Function**: If you need to iterate over a dictionary while keeping track of a counter, you can call the `enumerate()` function. The function returns an `enumerate` object, which assigns an integer to each item, like a counter. You can start the counter from any number, but by default, it starts from 0.

Assigning the index and item to separate loop variables is the common way to use `enumerate()`. For example, with `products.items()`, you can get the entire key-value pair in addition to the index:

```
for index, product in enumerate(products.items()):
    print(index, product)
```

Output:

```
0 ('Laptop', 990)
1 ('Smartphone', 600)
2 ('Tablet', 250)
3 ('Headphones', 70)
```

To customize the initial value of the count, you can pass a second argument to `enumerate()`. For example, here we are starting the count from 1.

```
for index, product in enumerate(products.items(), 1):
    print(index, product)
```

Output:

```
1 ('Laptop', 990)
2 ('Smartphone', 600)
3 ('Tablet', 250)
4 ('Headphones', 70)
```

Sets

- **Sets**: Sets are built-in data structures in Python that do not allow duplicate values. Sets are mutable and unordered, which means that their elements are not stored in any specific order, so you cannot use indices or keys to access them. Also, sets can only contain values of immutable data types, like numbers, strings, and tuples.
- **Defining a Set**: To define a set, you need to write its elements within curly brackets and separate them with commas.

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

- **Defining an Empty Set**: If you need to define an empty set, you must use the `set()` function. Only writing empty curly braces will automatically create a dictionary.

```
set() # Set
{}    # Dictionary
```

Common Set Methods

- **`add()` Method**: You can add an element to a set with the `add()` method, passing the new element as an argument.

```
my_set.add(6)
```

- **`remove()` and `discard()` Methods**: To remove an element from a set, you can either use the `remove()` method or the `discard()` method, passing the element you want to remove as an argument. The `remove()` method will raise a `KeyError` if the element is not found while the `discard()` method will not.

```
my_set.remove(4)
my_set.discard(4)
```

- **`clear()` method**: The `clear()` method removes all the elements from the set.

```
my_set.clear()
```

Mathematical Set Operations

- **`issubset()` and `issuperset()` Methods**: The `issubset()` and the `issuperset()` methods check if a set is a subset or superset of another set, respectively.

```
my_set = {1, 2, 3, 4, 5}
your_set = {2, 3, 4, 5}

print(your_set.issubset(my_set)) # True
print(my_set.issuperset(your_set)) # True
```

- **`isdisjoint()` Method**: The `isdisjoint()` method checks if two sets are disjoint, if they don't have elements in common.

```
print(my_set.isdisjoint(your_set)) # False
```

- **Union Operator (`|`)**: The union operator `|` returns a new set with all the elements from both sets.

```
my_set | your_set # {1, 2, 3, 4, 5, 6}
```

- **Intersection Operator (`&`)**: The intersection operator `&` returns a new set with only the elements that the sets have in common.

```
my_set & your_set # {2, 3, 4}
```

- **Difference Operator (`-`)**: The difference operator `-` returns a new set with the elements of the first set that are not in the other sets.

```
my_set - your_set # {1, 5}
```

- **Symmetric Difference Operator (`^`)**: The symmetric difference operator `^` returns a new set with the elements that are either on the first or the second set, but not both.

```
my_set ^ your_set # {1, 5, 6}
```

- **`in` Operator**: You can check if an element is in a set or not with the `in` operator.

```
print(5 in my_set)
```

Python Standard Library

- **Python Standard Library**: A library gives you pre-written and reusable code, like functions, classes, and data structures, that you can reuse in your projects. Python has an extensive standard library with built-in modules that implement standardized solutions for many problems and tasks. Some examples of popular built-in modules are `math`, `random`, `re` (short for "regular expressions"), and `datetime`.

Import Statement

- **Import Statement**: To access the elements defined in built-in modules, you use an import statement. Import statements are generally written at the top of the file. Import statements work the same for functions, classes, constants, variables, and any other elements defined in the module.
- **Basic Import Statement**: You can use the `import` keyword followed by the name of the module:

```
import module_name
```

Then, if you need to call a method from that module, you would use dot notation, with the name of the module followed by the name of the method.

```
module_name.method_name()
```

For example, you would write the following in your code to import the `math` module and get the square root of 36:

```
import math

math.sqrt(36)
```

- **Importing a Module with a Different Name**: If you need to import the module with a different name (also known as an "alias"), you can use `as` followed by the alias at the end of the import statement. This is often used for long module names or to avoid naming conflicts.

```
import module_name as module_alias
```

For example, to refer to the `math` module as `m` in your code, you can assign an alias like this:

```
import math as m
```

Then, you can access the elements of the module using the alias:

```
m.sqrt(36)
```

- **Importing Specific Elements**: If you don't need everything from a module, you can import specific elements using `from`. In this case, the import statement starts with `from`, followed by the module name, then the `import` keyword, and finally the names of the elements you want to import.

```
from module_name import name1, name2
```

Then, you can use these names without the module prefix in your Python script. For example:

```
from math import radians, sin, cos

angle_degrees = 40
angle_radians = radians(angle_degrees)

sine_value = sin(angle_radians)
cos_value = cos(angle_radians)

print(sine_value) # 0.6427876096865393
print(cos_value)  # 0.766044443118978
```

This is helpful, but it can result in naming conflicts if you already have functions or variables with the same name. Keep it in mind when choosing which type of import statement you want to use.

If you need to assign aliases to these names, you can do so as well, using the `as` keyword followed by the alias.

```
from module_name import name1 as alias1, name2 as alias2
```

- **Import Statement with Asterisk (`*`)**: The asterisk tells Python that you want to import everything in that module, but you want to import it so that you don't need to use the name of the module as a prefix.

```
from module_name import *
```

For example, if you this to import the `math` module, you'll be able to call any function defined in that module without specifying the name of the module as a prefix.

```
from math import *
print(sqrt(36))  # 6.0
```

However, this is generally discouraged because it can lead to namespace collisions and make it harder to know where names come from.

`if __name__ == '__main__'`

- **`__name__` Variable**: `__name__` is a special built-in variable in Python. When a Python file is executed directly, Python sets the value of this variable to the string `"__main__"`. But if the Python file is imported as a module into another Python script, the value of the `__name__` variable is set to the name of that module.

This is why you'll often find this conditional in Python scripts. It contains the code that you only want to run **only** if the Python script is running as the main program.

```
if __name__ == '__main__':
    # Code
```

Common Errors in Python

- **SyntaxError**: The error Python raises when your code does not follow its syntax rules. For example, the code `print("Hello there"` will lead to a syntax error with the message, `SyntaxError: '(' was never closed`, because the code is missing a closing parenthesis.
- **NameError**: Python raises a `NameError` when you try to access a variable or function you have not defined. For instance, if you have the line `print(username)` in your code without having a `username` variable defined first, you will get a name error with the message `NameError: name 'username' is not defined`.
- **TypeError**: This is the error Python throws when you perform an operation on two or more incompatible data types. For example, if you try to add a string to a number, you'll get the error `TypeError: can only concatenate str (not "int") to str`.
- **IndexError**: You'll get an `IndexError` if you access an index that does not exist in a list or other sequences like tuple and string. For example, in a `Hello world` string, the index of the last character is `11`. If you go ahead and access a character this way, `greet = "hello world"; print(greet[12])`, you'll get an error with the message `IndexError: string index out of range`.
- **AttributeError**: Python raises this error when you try to use a method or property that does not exist in an object of that type. For example, calling `.append()` on a string like `"hello".append("!")` will lead to an error with the message `AttributeError: 'str' object has no attribute 'append'`.

Good Debugging Techniques in Python

- **Using the `print` function**: Inserting `print()` statements around various points in your code while debugging helps you see the values of variables and how your code flows.
- **Using Python's Built-in Debugger (`pdb`)**: Python provides a `pdb` module for debugging. It's a part of the Python's standard library, so it's always available to use. With `pdb`, you can set a trace with the `set_trace()` method so you can start stepping through the code and inspect variables in an interactive way.
- **Leveraging IDE Debugging Tools**: Many integrated development environments (IDEs) and code editors like Pycharm and VS Code offer debugging tools with breakpoints, step execution, variable inspection, and other debugging features.

Exception Handling

- `try...except`: This is used to execute a block of code that might raise an exception. The `try` block is where you anticipate an error might occur, while the `except` block takes a specified exception and runs if that specified error is raised. Here's an example:

```
try:
  print(22 / 0)
except ZeroDivisionError:
  print('You can\'t divide by zero!')
  # You can't divide by zero!
```

You can also chain multiple `except` blocks so you can handle more types of exceptions:

```
try:
  number = int(input('Enter a number: '))
  print(22 / number)
except ZeroDivisionError:
  print('You cannot divide by zero!')
```

```
  # You cannot divide by zero! prints when you enter 0
except ValueError:
  print('Please enter a valid number!')
  # Please enter a valid number! prints when you enter a string
```

- **`else` and `finally`**: These blocks extend `try...except`. If no exception occurs, the `else` block runs. The `finally` block always runs regardless of errors.

```
try:
  result = 100 / 4
except ZeroDivisionError:
  print('You cannot divide by zero!') # This will not run
else:
  print(f'Result is {result}') # Result is 25.0
finally:
  print('Execution complete!') # Execution complete!
```

- **Exception Object**: This lets you access the exception itself for better debugging and printing the direct error message. To access the exception object, you need to use the `as` keyword. Here's an example:

```
try:
    value = int('This will raise an error')
except ValueError as e:
    print(f'Caught an error: {e}')
    # Caught an error: invalid literal for int() with base 10: 'This will raise an error'
```

- **The `raise` Statement**: This allows you to manually raise an exception. You can use it to throw an exception when a certain condition is met. Here's an example:

```
def divide(a, b):
    if b == 0:
        raise ZeroDivisionError('You cannot divide by zero')
    return a / b
```

Exception Signaling

The `raise` statement is also useful when you create your own custom exceptions, as you can use it to throw an exception with a custom message. Here's an example of that:

```
class InvalidCredentialsError(Exception):
    def __init__(self, message="Invalid username or password"):
        self.message = message
        super().__init__(self.message)
```

```
def login(username, password):
    stored_username = "admin"
    stored_password = "password123"

    if username != stored_username or password != stored_password:
        raise InvalidCredentialsError()

    return f"Welcome, {username}!"
```

Here's a how you can use the `login` function from the `InvalidCredentialsError` exception:

```
# failed login attempt
try:
    message = login("user", "wrongpassword")
    print(message)
except InvalidCredentialsError as e:
    print(f"Login failed: {e}")

# successful login attempt
try:
    message = login("admin", "password123")
    print(message)
except InvalidCredentialsError as e:
    # This block is not executed because the login was successful
    print(f"Login failed: {e}")
else:
    # The else block runs if the 'try' block completes without an exception
    print(message)
```

The `raise` statement can also be used with the `from` keyword to chain exceptions, showing the relationship between different errors:

```
def parse_config(filename):
  try:
      with open(filename, 'r') as file:
          data = file.read()
          return int(data)
  except FileNotFoundError:
      raise ValueError('Configuration file is missing') from None
  except ValueError as e:
      raise ValueError('Invalid configuration format') from e
```

```
config = parse_config('config.txt')
```

Python Classes and Objects

- **Class Definition**: A class is a blueprint for creating objects. It defines the behavior an object will have through its attributes and methods. Here is a basic example of a class definition in Python:

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print(f'{self.name.upper()} says woof woof!')
```

- **Creating Objects**: Objects are instances of a class. They are created by calling the class with the necessary arguments.

```
dog1 = Dog('Jack', 3)
dog2 = Dog('Thatcher', 5)

dog1.bark()  # JACK says woof woof!
dog2.bark()  # THATCHER says woof woof!
```

- **Calling methods with objects**: You can call methods on objects to perform actions or retrieve information.

```
objectName1.methodName()
objectName2.methodName()
```

- **Difference Between Class and Object**: A class is a reusable template, while an object is a specific instance of that class with actual data.

Attributes

- **Instance Attributes**: Defined in `__init__()` using `self`, and unique to each object.
- **Class Attributes**: Defined directly inside the class and shared by all instances.

```
class Dog:
    species = 'French Bulldog'  # Class attribute

    def __init__(self, name):
        self.name = name  # Instance attribute

print(Dog.species) # French Bulldog
```

```
jack = Dog('Jack')
print(jack.name)     # Jack
print(jack.species)  # French Bulldog
```

Methods

- **Methods**: Functions defined inside a class that operate on the object's attributes.

```
class Car:
    def __init__(self, color, model):
        self.color = color
        self.model = model

    def describe(self):
        return f'This car is a {self.color} {self.model}'

my_car_1 = Car('red', 'Tesla Model S')
print(my_car_1.describe())  # This car is a red Tesla Model S
```

- **Accessing Methods**: Call methods on objects using the dot notation. Here is an example of calling the `describe` method on two different car objects:

```
class Car:
 def __init__(self, color, model):
   self.color = color
   self.model = model

 def describe(self):
   return f'This car is a {self.color} {self.model}'

my_car_1 = Car('red', 'Tesla Model S')
my_car_2 = Car('green', 'Lamborghini Revuelto')

print(my_car_1.describe()) # Calling methods using the dot notation

print(my_car_2.describe()) # Calling methods using the dot notation
```

Dunder (Magic) Methods

- **Definition**: Special methods that start and end with a double underscore (e.g., `__init__`, `__len__`, `__str__`, `__eq__`). Python uses them internally for built-in operations.

```
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    def __len__(self):
        return self.pages

    def __str__(self):
        return f"'{self.title}' has {self.pages} pages"

    def __eq__(self, other):
        return self.pages == other.pages

book1 = Book('Built Wealth Like a Boss', 420)
print(len(book1))        # 420
print(str(book1))        # 'Built Wealth Like a Boss' has 420 pages
```

- **Calling dunder methods indirectly**: You don't need to call dunder methods directly. Instead, Python automatically calls them when certain actions happen. These operations include:
    - **arithmetic operations like addition, subtraction, multiplication, division, and others**. In addition, `__add__()` is called, `__sub__()` for subtraction, `__mul__()` for multiplication, and `__truediv__()` for division.
    - **string operations like concatenation, repetition, formatting, and conversion to text**. `__add__()` is called for concatenation, `__mul__()` for repetition, `__format__()` for formatting, `__str__()` and `__repr__()` for text conversion, and so on.
    - **comparison operations like equality, less-than, greater-than, and others**. `__eq__()` is called for equality checks, `__lt__()` for less-than, `__gt__()` for greater-than, and so on.
    - **iteration operations like making an object iterable and advancing through items**. `__iter__()` is called to return an iterator and `__next__()` to fetch the next item.

Real World Example: Shopping Cart

- **Cart Class with Dunder Methods**: Allows adding, removing, iterating, and checking contents with built-in behavior.

```
class Cart:
    def __init__(self):
        self.items = []

    def add(self, item):
        self.items.append(item)

    def remove(self, item):
        if item in self.items:
            self.items.remove(item)
```

```
        else:
            print(f'{item} is not in cart')

    def list_items(self):
        return self.items

    def __len__(self):
        return len(self.items)

    def __getitem__(self, index):
        return self.items[index]

    def __contains__(self, item):
        return item in self.items

    def __iter__(self):
        return iter(self.items)

cart = Cart()
cart.add('Laptop')
print(len(cart))        # 1
print('Laptop' in cart) # True
```

What is Object-Oriented Programming?

- **Object-oriented programming**: A programming style in which developers treat everything in their code like a real-world object. It is popularly called OOP. The four key principles that help you organize and manage code effectively are **encapsulation**, **inheritance**, **polymorphism**, and **abstraction**
- **Classes**: The blueprint for creating objects. Every single object created from a class has attributes that define data and methods that determine the behaviors of the objects.

What is Encapsulation?

- **Encapsulation**: The bundling of the attributes and methods of an object into a single unit. It lets you hide the internal state of the object behind a simple set of public methods and attributes that act like doors. Behind those doors are private attributes and methods that control how the data changes and who can see it.
- **Example of Encapsulation**: If you want to track a wallet balance, you will allow deposit and withdrawal, but you won't want anyone to tamper with the wallet balance itself:

```
class Wallet:
   def __init__(self, balance):
       self.__balance = balance # Private attribute

   def deposit(self, amount):
       if amount > 0:
           self.__balance += amount # Add to the balance safely
```

```
    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount # Remove from the balance safely

account = Wallet(500)
print(account.__balance) # AttributeError: 'Wallet' object has no attribute '__balance'
```

- **Difference Between Prefixing Attributes with Single and Double Underscore**: Prefixing attributes and methods with a single underscore means they are meant for internal use. This is a convention, and it doesn't enforce accessing attributes from the outside. Prefixing attributes and methods with a double underscore effectively prevents them from being accessed from outside of their class.

What Are Getters and Setters?

- **Getters and Setters**: Methods that let you control how the attributes of a class are accessed and modified. You retrieve values with getters and you set values with setters.
- **Properties**: They connect getters and setters, and allow access to data. They run extra logic behind the scenes when you get, set, or delete values.
- **Why Properties Instead of Methods**: Properties are used instead of methods for better readability and cleaner code. They let you access values with dot notation, like regular attributes, without parentheses.
- **Creating a Getter**: To create a getter, you use the `@property` decorator. Here's a getter that gets the radius of a circle:

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self): # A getter to get the radius
        return self._radius

    @property
    def area(self):  # A getter to calculate area
        return 3.14 * (self._radius ** 2)

my_circle = Circle(3)

print(my_circle.radius) # 3
print(my_circle.area) # 28.26
```

- **Creating a Setter**: To create the setter that will set the radius, you have to define another method with the same name and use `@<property_name>.setter` above it:

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
```

```
    def radius(self):  # A getter to get the radius
        return self._radius

    @radius.setter
    def radius(self, value):  # A setter to set the radius
        if value <= 0:
            raise ValueError('Radius must be positive')
        self._radius = value

my_circle = Circle(3)
print('Initial radius:', my_circle.radius) # Initial radius: 3

my_circle.radius = 8
print('After modifying the radius:', my_circle.radius) # After modifying the radius: 8
```

- **How Python Handles Getters and Setters**: Once you define getters and setters, Python automatically calls them under the hood whenever you use normal attribute syntax this way:

```
my_circle.radius # This will call the getter
my_circle.radius = 4 # This will call the setter
```

When setting a value, you should not assign to the property name itself because that will cause a `RecursionError`. Use a separate internal name, often with an underscore, to store the value.

- **Deleter**: After setting and getting a value with setter and getter, you can control how it is deleted with a `deleter`. A deleter runs custom logic when you use the `del` statement on a property. To create a deleter, you use the `@<property_name>.deleter` decorator.

```
  # Deleter
    @radius.deleter
    def radius(self):
        print("Deleting radius...")
        del self._radius
```

What Is Inheritance and How Does It Promote Code Reuse?

- **Inheritance**: The process by which a child class uses the attributes and methods of a parent class. Inheritance promotes code reuse, provides clear hierarchies, and customizes behavior without rewriting everything. To implement inheritance, a child class takes in the name of a parent class:

```
class Parent:
    # Parent attributes and methods

class Child(Parent):
    # Child inherits, extends, and/or overrides where necessary
```

- **Single and Multiple Inheritance**: When a child class inherits properties and methods from a single parent, as you can see above, the process is called **single inheritance**. When a child class inherits properties and methods from more than one parent, that is **multiple inheritance**. Here's the syntax for that:

```
class Parent:
    # Attributes and methods for Parent

class Child:
    # Attributes and methods for Child

class GrandChild(Parent, Child):
    # GrandChild inherits from both Parent and Child
    # GrandChild can combine or override behavior from each
```

- `super()` **Function**: A function that lets you override a method from a parent inside a child class.

What Is Polymorphism and How Does It Promote Code Reuse?

- **Polymorphism**: The OOP principle that lets different classes use the same method name, but each class implements it differently when called. Here's the syntax for it:

```
class A:
   def action(self): ...

class B:
   def action(self): ...

class C:
   def action(self): ...

Class().method()  # Works for A, B, or C
```

- **Inheritance-based polymorphism**: A parent sets up a method, and each child class twists it to their use.

What is Name Mangling and How Does it Work?

- **Name Mangling**: A process in which Python internally renames an attribute prefixed with a double underscore by adding an underscore and the class name as a prefix, turning `__attribute` into `_ClassName__attribute`.
- **The Purpose of Name Mangling**: The main purpose of name mangling is to prevent accidental attribute and method overriding when you use inheritance. Here's a code that makes that more understandable:

```
class Parent:
    def __init__(self):
        self.__data = 'Parent data'

class Child(Parent):
```

```
    def __init__(self):
        super().__init__()
        self.__data = 'Child data'

c = Child()
print(c.__dict__) # {'_Parent__data': 'Parent data', '_Child__data': 'Child data'}
```

What Is Abstraction and How Does It Help Keep Complex Systems Organized?

- **Abstraction**: A programming concept in which complex implementation details of object or system are hidden and only the essential features are shown. In Python and other programming languages, abstraction simplifies complex systems by increasing reusability.
- **Example of Abstraction**: A good example of abstraction in everyday life is a car letting you just use the wheel, pedals, and shifter without knowing how the engine or brakes work.
- **How Python Implements Abstraction**: Python implements abstraction through the `abc` module. The module provides the `ABC` class (abstract base class) and the `@abstractmethod` decorator. An abstract base class (ABC) defines the common methods and properties subclasses must implement. It can't be instantiated.
- **How Abstract Method is Defined**: An abstract method is defined with `@abstractmethod` and must be overridden in subclasses, even if it has a default implementation. The basic syntax of abstraction looks like this:

```
from abc import ABC, abstractmethod

# Define an abstract base class
class AbstractClass(ABC):
    @abstractmethod
    def abstract_method(self):
        pass

# Concrete subclass that implements the abstract method
class ConcreteClassOne(AbstractClass):
    def abstract_method(self):
        print('Implementation in ConcreteClassOne')

# Another concrete subclass
class ConcreteClassTwo(AbstractClass):
    def abstract_method(self):
        print('Implementation in ConcreteClassTwo')
```

Algorithms and Big O Notation

- **Algorithms**: A set of unambiguous instructions for solving a problem or carrying out a task. Algorithms must finish in a finite number of steps and each step must be precise and unambiguous.
- **Big O Notation**: Describes the worst-case performance, or growth rate, of an algorithm as the input size increases. It focuses on how resource usage grows with input size, ignoring constant factors and lower-order terms.

Common Time Complexities

- **O(1) - Constant Time**: Algorithm takes the same amount of time regardless of input size.

```
def check_even_or_odd(number):
    if number % 2 == 0:
        return 'Even'
    else:
        return 'Odd'
```

- **O(log n) - Logarithmic Time**: Time increases slowly as input grows. Common in algorithms that repeatedly reduce problem size by a fraction (like Binary Search).
- **O(n) - Linear Time**: Running time increases proportionally to input size.

```
for grade in grades:
    print(grade)
```

- **O(n log n) - Log-Linear Time**: Common time complexity of efficient sorting algorithms like Merge Sort and Quick Sort.
- **O(n^2) - Quadratic Time**: Running time increases quadratically. Often seen in nested loops.

```
for i in range(n):
    for j in range(n):
        print("Hello, World!")
```

Space Complexity

- **O(1) - Constant Space**: Algorithm uses same amount of memory regardless of input size.
- **O(n) - Linear Space**: Memory usage grows proportionally with input size.
- **O(n^2) - Quadratic Space**: Memory usage grows quadratically with input size.

Problem-Solving Techniques

- **Understanding the Problem**: Read the problem statement multiple times. Identify the input, expected output, and how to transform input to output.
- **Pseudocode**: High-level description of algorithm logic that is language-independent. Uses common written language mixed with programming constructs like `IF`, `ELSE`, `FOR`, `WHILE`.

```
GET original_string
SET reversed_string = ""
FOR EACH character IN original_string:
  ADD character TO THE BEGINNING OF reversed_string
DISPLAY reversed_string
```

- **Edge Cases**: Specific, valid inputs that occur at the boundaries of what an algorithm should handle. Always consider and test edge cases.

Arrays

- **Static Arrays**: Have a fixed size determined at initialization. Elements stored in adjacent memory locations. Size cannot be changed during program execution.

- **Dynamic Arrays**: Can grow or shrink automatically during program execution. Handle resizing through automatic copying to larger arrays when needed.

Python Lists (Dynamic Arrays)

```
numbers = [3, 4, 5, 6]

# Access elements
numbers[0]  # 3

# Update elements
numbers[2] = 16

# Add elements
numbers.append(7)
numbers.insert(3, 15)  # Insert at specific index

# Remove elements
numbers.pop(2)  # Remove at specific index
numbers.pop()   # Remove last element
```

Time Complexities for Dynamic Arrays

- **Access**: O(1)
- **Insert at end**: O(1) average, O(n) when resizing needed
- **Insert in middle**: O(n)
- **Delete**: O(n) for middle, O(1) for end

Stacks

- **Stacks**: Last-In, First-Out (LIFO) data structure. Elements added and removed from the top only.
- **Push Operation**: Adding an element to the top of the stack. Time complexity: O(1).
- **Pop Operation**: Removing an element from the top of the stack. Time complexity: O(1).

```
# Using Python list as stack
stack = []

# Push operations
stack.append(1)
stack.append(2)
stack.append(3)

# Pop operations
top_element = stack.pop()  # Returns 3
```

Queues

- **Queues**: First-In, First-Out (FIFO) data structure. Elements added to the back and removed from the front.
- **Enqueue Operation**: Adding an element to the back of the queue. Time complexity: O(1).
- **Dequeue Operation**: Removing an element from the front of the queue. Time complexity: O(1).

```
from collections import deque

# Using deque for efficient queue operations
queue = deque()

# Enqueue operations
queue.append(1)
queue.append(2)
queue.append(3)

# Dequeue operations
first_element = queue.popleft()  # Returns 1
```

Linked Lists

- **Linked Lists**: Linear data structure where each node contains data and a reference to the next node. Nodes are connected like a chain.

Singly Linked Lists

- **Structure**: Each node has data and one reference to the next node.
- **Traversal**: Can only move forward from head to tail.
- **Head Node**: First node in the list, usually the only directly accessible node.
- **Tail Node**: Last node in the list, points to `None`.

Operations and Time Complexities

- **Insert at beginning**: O(1)
- **Insert at end**: O(n) - must traverse to end
- **Insert in middle**: O(n) - must traverse to position
- **Delete from beginning**: O(1)
- **Delete from end**: O(n) - must traverse to find previous node
- **Delete from middle**: O(n) - must traverse to find node

Doubly Linked Lists

- **Structure**: Each node has data and two references: next node and previous node.
- **Traversal**: Can move in both directions.
- **Memory**: Requires more memory than singly linked lists due to extra reference.

Hash Maps and Sets

Maps and Hash Maps

- **Map (Abstract Data Type)**: Manages collections of key-value pairs. Every key must be unique, but values can be repeated.
- **Hash Map**: Concrete implementation of map ADT using hashing technique. Uses hash function to generate hash values for keys, which determine storage location in underlying array.

Python Dictionaries (Hash Maps)

```
# Creating dictionaries
my_dictionary = {
    "A": 1,
    "B": 2,
    "C": 3
}

# Alternative creation
my_dictionary = dict(A=1, B=2, C=3)

# Access and modify
value = my_dictionary["A"]  # 1
my_dictionary["A"] = 4      # Update value
del my_dictionary["A"]      # Remove key-value pair

# Check membership
"C" in my_dictionary

# Get keys, values, items
my_dictionary.keys()
my_dictionary.values()
my_dictionary.items()
```

Time Complexities for Hash Maps

- **Average case**: O(1) for insert, get, delete
- **Worst case**: O(n) when many hash collisions occur

Sets

- **Sets**: Unordered collections of unique elements. No duplicates allowed, no specific order maintained.
- **Immutable Elements Only**: Sets can only contain immutable data types (numbers, strings, tuples) because hash values must remain constant.

```
# Creating sets
numbers = {1, 2, 3, 4}
empty_set = set()  # Must use set(), not {}

# Add and remove elements
```

```
numbers.add(5)
numbers.remove(4)      # Raises KeyError if not found
numbers.discard(4)     # No error if not found

# Set operations
set_a = {1, 2, 3, 4}
set_b = {2, 3, 4, 5, 6}

# Union, intersection, difference, symmetric difference
set_a.union(set_b)                  # or set_a | set_b
set_a.intersection(set_b)           # or set_a & set_b
set_a.difference(set_b)             # or set_a - set_b
set_a.symmetric_difference(set_b)   # or set_a ^ set_b

# Subset and superset checks
set_a.issubset(set_b)
set_a.issuperset(set_b)
set_a.isdisjoint(set_b)

# Membership testing
5 in numbers
```

Time Complexities for Sets

- **Average case**: O(1) for add, remove, membership testing
- **Worst case**: O(n) due to hash collisions

Hash Collisions

- **Hash Collision**: Occurs when two different keys produce the same hash value.
- **Collision Resolution Strategies**:
  - **Chaining**: Each array index points to a linked list storing all elements with same hash value
  - **Open Addressing**: Search for next available index using predefined sequence

When to Use Each Data Structure

- **Lists**: When you need ordered, indexed access and don't know size in advance
- **Stacks**: For LIFO operations (undo functionality, expression evaluation, backtracking)
- **Queues**: For FIFO operations (task scheduling, breadth-first search)
- **Linked Lists**: When frequent insertion/deletion at beginning, unknown size, no random access needed
- **Hash Maps**: For fast key-value lookups, counting occurrences, caching
- **Sets**: For uniqueness checking, mathematical set operations, removing duplicates

Searching Algorithms

Searching algorithms let you search for a target within a certain list of items.

In computer science, there are two searching algorithms you should know about. They are **linear search** and **binary search** algorithms. It is important to understand the differences between the two algorithms and when to use each one.

Linear Search

- Linear search iterates through a list of items, checking each item from the beginning until the target item is found.
- If the target item is found, the index where it is located in the list is returned.
- If the target is not found, it returns `-1`, which means **invalid index** in most programming languages.
- Because linear search checks each item until it finds the target, it is not efficient for a large list of items.
- The time complexity of linear search is `O(n)` because the time it takes to search through the list grows linearly with the size of the list.
- The space complexity of linear search is `O(1)` because it doesn't require any additional space to search through the list.

Binary Search

- Binary search works by dividing a list of items in half, and checking if the target value is in the middle of the list.
- The condition for binary search to work is that the items in the list must be in ascending order.
- Binary search is a more efficient algorithm for searching through a large list of items because it divides the list of items in half and ignores any half where the target is not found.
- If the target item is found in the middle of the list, the index of the target item is returned.
- If the item is not found, the algorithm checks if the target item is in the left or right half of the list.
- It continues to divide the remaining parts of the list into halves until the target item is found.
- If the target item is finally not found in the list, it returns `-1`
- The time complexity of binary search is `O(log n)` because the time it takes to search through the list grows logarithmically with the size of the list.
- The space complexity of binary search is `O(1)` because it doesn't require any additional space to search through the list.

How Linear Search Differs from Binary Search

- Binary search is more suitable for a large list of items compared to linear search.
- The time complexity of linear search is `O(n)` because the time it takes to search through the list grows linearly with the size of the list.
- The time complexity of binary search is `O(log n)` because the time it takes to search through the list grows logarithmically with the size of the list.

Sorting Algorithms and Divide-and-Conquer

In computer science, divide-and-conquer is a technique used to break down a problem into smaller sub-problems so they are easier to solve. Recursion is the technique often employed in divide-and-conquer, and divide-and-conquer is a powerful strategy used to implement many efficient sorting algorithms like merge sort.

Merge Sort

- Merge sort is a sorting algorithm that follows the divide-and-conquer approach.
- It works by recursively dividing a list into smaller sub-lists until each sub-list contains only one element.
- It then repeatedly merges the sub-lists back together in a sorted order.
- The time complexity for merge sort is `O(n log n)` because the list is continuously divided in half `(log n)` and then merged together `(O(n))`.
- The space complexity of merge sort is `O(n)` because it is not an in-place sorting algorithm.

Graphs Overview

A graph is a set of nodes (vertices) connected by edges (connections). Each node can connect to multiple other nodes, forming a network. The different types of graphs include:

- Directed: edges have a direction (from one node to another), often represented with straight lines and arrows.
- Undirected: edges have no direction, represented with simple lines.
- Vertex: each node is associated to a label or identifier.
- Cyclic: contains cycles (a path that starts and ends at the same node).
- Acyclic (DAG): does not contain cycles.
- Edge labeled: each edge has a label usually drawn next to corresponding edge.
- Weighted: edges have weights (values) associated with them, that can be used to perform arithmetic operations.
- Disconnected: contains two or more nodes that are not connected by any edges.

Graphs are used in various applications such as maps, networks, recommendation systems, dependency resolution.

Graph Traversals

This involves visiting all the nodes in a graph. The two main algorithms are:

- **Breadth-First Search (BFS)**
  - Uses a queue.
  - Explores level by level.
  - Finds shortest path in unweighted graphs.
- **Depth-First Search (DFS)**
  - Uses a stack (or recursion).
  - Explores a branch fully before backtracking.
  - Useful for cycle detection and path finding.

Graph Representations

Graphs can be represented in two main ways:

- **Adjacency List**
  - Each node has a list of its neighbors.
  - Space efficient for sparse graphs.
  - Easy to iterate over neighbors.
- **Adjacency Matrix**
  - A 2D array where rows and columns represent nodes.
  - Space intensive for large graphs.
  - Fast to check if an edge exists between two nodes.

Trees

A tree is a special type of graph that is acyclic and connected. Key properties include:

- They have no loops or cycles (paths where the start and end nodes are the same).
- They must be connected (every node can be reached from every other node).

Common types of trees

The most common types of trees are:

- Binary Trees
  - Each node has at most two children, a left and a right child.
- Binary Search Trees (BST)
  - A binary tree in which every left child is less than its parent, and every right child is greater than its parent.

Tries

Also known as prefix trees, they are used to store sets of strings, where each node represents a character.

Shared prefixes are stored only once, making them efficient for tasks like autocomplete and spell checking.

Search and insertion operations have a time complexity of O(L), where L is the length of the string.

Priority Queues

A priority queue is an abstract data type where each element has a priority.

Queues and stacks consider only the order of insertion, while priority queues consider the priority of elements.

Standard queues follow FIFO (First In First Out) and stacks follow LIFO (Last In First Out). However, in a priority queue, elements with higher priority are served before those with lower priority, regardless of their insertion order.

Heaps

It's a specialized tree-based data structure with a very specific property called the heap property.

The heap property determines the relationship between parent and child nodes. There are two types of heaps:

- Max-Heap
  - The value of each parent node is greater than or equal to the values of its children.
  - The largest element is at the root.
- Min-Heap
  - The value of each parent node is less than or equal to the values of its children.
  - The smallest element is at the root.

Python `heap` module example

```
import heapq
```

```python
# Create empty heap
my_heap = []

# Insert elements
heapq.heappush(my_heap, 9)
heapq.heappush(my_heap, 3)
heapq.heappush(my_heap, 5)

# Remove smallest element
print(heapq.heappop(my_heap))  # 3

# Push + Pop in one step
print(heapq.heappushpop(my_heap, 2)) # 2

# Transform list into heap
nums = [5, 7, 3, 1]
heapq.heapify(nums)
```

Using Priorities

```python
my_heap = []
heapq.heappush(my_heap, (3, "A"))
heapq.heappush(my_heap, (2, "B"))
heapq.heappush(my_heap, (1, "C"))

# Removes lowest number = highest priority
print(heapq.heappop(my_heap))  # (1, "C")
```

Introduction to Dynamic Programming

- **Definition**: Dynamic programming is an algorithmic technique that solves complex problems by breaking them down into simpler subproblems and storing the results to avoid redundant calculations.
- **Overlapping Subproblems**: The same smaller problems appear multiple times when solving the larger problem. Instead of recalculating these subproblems repeatedly, we store their solutions.
- **Optimal Substructure**: The optimal solution to the problem contains optimal solutions to its subproblems. This means we can build up the best solution by combining the best solutions to smaller parts.

Dynamic Programming Solutions

- **Memoization (Top-Down Approach)**: Memoization stores the results of expensive function calls and returns the cached result when the same inputs occur again.

```python
def climb_stairs_memo(n, memo={}):
    """Dynamic programming with memoization"""
```

```
    # Check if we've already calculated this value
    if n in memo:
        return memo[n]  # Return cached result - O(1) lookup!

    # Base cases
    if n <= 2:
        return n

    # Calculate once and store in memo for future use
    memo[n] = climb_stairs_memo(n-1, memo) + climb_stairs_memo(n-2, memo)
    return memo[n]
```

- **Tabulation (Bottom-Up Approach)**: Tabulation builds the solution from the ground up, filling a table with solutions to subproblems.

```
def climb_stairs_tabulation(n):
    """Dynamic programming with tabulation"""
    if n <= 2:
        return n

    # Create array to store results for all steps from 0 to n
    dp = [0] * (n + 1)
    dp[1] = 1  # 1 way to reach step 1
    dp[2] = 2  # 2 ways to reach step 2

    # Build up the solution iteratively
    for i in range(3, n + 1):
        # Ways to reach step i = ways to reach (i-1) + ways to reach (i-2)
        dp[i] = dp[i-1] + dp[i-2]

    return dp[n]
```

Real-World Applications Using Dynamic Programming

- **Route Optimization**: GPS systems use dynamic programming algorithms to find shortest paths between locations.
- **Text Processing**: Spell checkers and autocomplete features often rely on dynamic programming to calculate edit distances between words.
- **Financial Modeling**: Investment strategies and portfolio optimization frequently employ dynamic programming techniques.
- **Resource Allocation**: The knapsack problem and its variants appear in scheduling, budgeting, and resource management.

When to Use Dynamic Programming

You should consider using dynamic programming in the following scenarios:

- The problem can be broken down into overlapping subproblems.
- The problem exhibits optimal substructure.

- A naive recursive solution would involve repeated calculations.
- You need to optimize for time complexity at the cost of space complexity.