


Core tenets of python syntax:

Python Built-in functions:

1. Creating and Manipulating Lists

- `list()` : Creates a new list. You can turn other iterables (like strings, tuples, sets) into lists.


python

 Copy code

```
my_list = list("abc") # Creates ['a', 'b', 'c']
```

- `append()` : Adds an item to the **end** of the list.


python

 Copy code

```
my_list.append(4)
```

- `extend()` : Adds multiple items from an iterable (like another list) to the end of the list.


python

 Copy code

```
my_list.extend([5, 6]) # Adds 5 and 6 to the list
```

- `insert(index, item)` : Inserts an item at a specified position in the list.

python

 Copy code


```
my_list.insert(1, "x") # Inserts "x" at index 1
```

Python Built-in functions:

2. Accessing List Elements

- `index()` : Returns the index of the first occurrence of an item in the list.


python

 Copy code

```
my_list.index("a") # Returns the index of "a"
```

- `count()` : Counts the number of times an item appears in the list.


python

 Copy code

```
my_list.count("a") # Counts occurrences of "a"
```

- `slice()` : Lets you access a subset of a list by specifying a start, stop, and step.

python

 Copy code


```
my_list[1:4] # Slices the list from index 1 to 3
```

Python Built-in functions:

3. Modifying and Removing Elements

- `remove()` : Removes the first occurrence of a specific item from the list.

python

 Copy code

```
my_list.remove("a") # Removes "a" from the list
```

- `pop()` : Removes and returns the item at a given index. If no index is specified, it removes the last item.

python

 Copy code

```
my_list.pop() # Removes and returns the last item
```

- `clear()` : Empties all items from the list.

python

 Copy code

```
my_list.clear() # Clears all items
```

4. Sorting and Reversing

- `sort()` : Sorts the list **in place** (i.e., modifies the original list) in ascending order by default.


python

 Copy code

```
my_list.sort() # Sorts my_list
```

- `sorted()` : Returns a **new sorted list** without changing the original list.

python

 Copy code

```
sorted_list = sorted(my_list) # Creates a sorted version of my_list
```

- `reverse()` : Reverses the list **in place**.

python

 Copy code


```
my_list.reverse() # Reverses the order of my_list
```

Python Built-in functions:

5. Looping and Filtering

- `next()` : Used to get the next item from an iterator or generator.


python

 Copy code

```
iterator = iter([1, 2, 3])
next(iterator) # Returns 1, then 2, then 3 on each call
```

- `filter()` : Creates an iterator with items that meet a specified condition.


python

 Copy code

```
filtered = filter(lambda x: x > 1, my_list) # Filters items greater than 1
```

- `range()` : Creates a sequence of numbers, useful in loops.

python

 Copy code


```
for i in range(5): # Loops from 0 to 4
    print(i)
```

Python Built-in functions:

6. Random Selection

- `choice()`: Randomly selects a single item from a list or any sequence.

python

 Copy code

```
import random
random.choice(my_list) # Randomly selects an item from my_list
```

7. Handling Sets (for `add()`)

- `add()`: Adds an item to a set. This function is only for sets, not lists.

python


 Copy code

```
my_set = {1, 2}
my_set.add(3) # Adds 3 to the set, if it isn't already present
```

8. Other Keywords and Special Uses

- `pass`: A placeholder keyword used when a statement is required syntactically but no action is needed.

python

 Copy code

```
def my_function():
    pass # Does nothing, useful for stubs or incomplete code
```

Summary Table

Function	Purpose
<code>list()</code>	Create a new list
<code>append()</code>	Add an item to the end of a list
<code>extend()</code>	Add multiple items to a list
<code>index()</code>	Find the position of an item
<code>count()</code>	Count occurrences of an item
<code>slice()</code>	Get a sub-list
<code>remove()</code>	Remove first occurrence of an item
<code>pop()</code>	Remove item at index (default last)
<code>clear()</code>	Empty the list
<code>sort()</code>	Sort list in place
<code>sorted()</code>	Return a new sorted list
<code>reverse()</code>	Reverse list in place
<code>next()</code>	Get the next item from an iterator
<code>filter()</code>	Filter items based on a condition
<code>range()</code>	Create a sequence of numbers
<code>choice()</code>	Randomly select an item from a list
<code>add()</code>	Add item to a set
<code>pass</code>	Do nothing (placeholder)

Lists[] vs Tuples():

1. Mutability:

List: Mutable, meaning you can change its contents (e.g., add, remove, or modify items).

Tuple: Immutable, meaning once it's created, you can't change its contents.

When to use:

Use a **list** when you expect the data to change. For example, a list of users on a website, which could grow or shrink over time.

Use a **tuple** for data that should remain constant throughout the program, such as coordinates of a point or fixed settings.

2. Intended Use:

List: Best for sequences of items that might change. Lists are like arrays in other languages and are ideal for collections of related items.

Tuple: Ideal for structured, fixed-size data. It can serve as a lightweight, read-only record for simple, fixed structures.

Examples:

List: Shopping cart items, to-do lists, or a series of temperature readings that may change.

Tuple: An (x, y) coordinate, RGB color values **(255, 255, 255)**, or a date **(year, month, day)**.

3. Data Integrity:

Tuple: Because tuples are immutable, they help prevent accidental changes. This makes them useful for "write-once, read-many" data.

List: Lists, being mutable, are at risk of unintended changes if passed around the code.

When to use:

For data that should stay safe and unaltered, choose a **tuple**. If you pass this data to multiple functions, its integrity remains intact.

For data that will evolve or be updated frequently, a **list** is better suited.

4. Function Arguments:

Tuple: Often used for passing fixed sets of items to functions, especially when the function expects the data structure to stay unchanged.

List: Used when passing variable sets of items to functions, especially when the function might modify the data.

5. When to use **list()** vs **[]**:

Use **[]** for simplicity and readability when initializing an empty list or creating a list with values, like `my_list = []` or `my_list = [1, 2, 3]`.


Use **list()** if you're converting an iterable to a list (e.g., converting a tuple `my_list = list((1, 2, 3))`) or if it's more readable in your specific context.

Quick Summary

Use **lists** for collections of items that may change.

Use **tuples** for fixed, unchanging data or where data immutability is essential.

python

 Copy code

```
# List example - shopping cart that can change
shopping_cart = ["apple", "banana", "orange"]
shopping_cart.append("grape") # Lists are mutable

# Tuple example - coordinates that stay the same
origin = (0, 0) # Tuples are immutable, ideal for fixed data
```


When to Use `return` vs `print()`:

`print()`:

When you just want to display information to the user, and you don't need to use it further in your code.

`print` is for showing information directly on the screen (usually for debugging or user interaction). If you just need to show the result to the user, print is the right choice.


When you're debugging or inspecting the values during development.

While developing, you might use `print` to display the values of variables or intermediate results to help understand what's going wrong. This helps with debugging but is usually removed once your program is final.

When you are working with a specific task like logging or generating reports.

If the purpose of the function is just to display results or generate output (for example, for printing reports), print is suitable.

python

 Copy code

```
def greet(name):  
    print(f"Hello, {name}!") # Directly print a greeting to the screen  
  
greet("Alice") # This will print the greeting to the console
```

return

When the function is performing a computation or processing data that needs to be used later.

If the result of the function needs to be used by other parts of your program (like storing it in a variable, passing it to another function, or using it in calculations), you should use **return**.

When you need to pass data back to the caller for further use or manipulation.

return gives the calling code access to the result, so it can be stored, processed, or logged later. For example, you might return a value from a mathematical calculation or return a string for further processing.

When you need to make a function reusable.

return allows the function to be reusable in different contexts, whereas **print** directly outputs the result to the screen, which limits its use.

```
python Copy code  
  
def add(a, b):  
    return a + b # This returns the sum of a and b so it can be used elsewhere  
  
result = add(3, 4) # Store the result in a variable  
print(result) # Later, print the result
```

Best Practices:

- **Return for logic and data:** If the purpose of your function is to compute or manipulate data, use **return**. This makes the function flexible and reusable.
- **Print for user-facing output:** If you need to show something to the user immediately (like status updates, results, or messages), use **print**.

Writing functions effectively:

1. Single Responsibility Principle (SRP):

A function should do one thing and do it well.

The function should perform a specific task or calculation and not try to handle multiple responsibilities at once.

Example: A function to calculate the area of a circle should only calculate the area and not print the result or modify other values.

Good Example:

python

 Copy code

```
def calculate_area(radius):  
    return 3.14 * (radius ** 2)
```

Bad Example:

python

 Copy code

```
def calculate_area_and_print(radius):  
    area = 3.14 * (radius ** 2)  
    print(f"The area is {area}")  
    return area # Too many responsibilities: calculation and printing
```

2. Use Parameters Instead of Global Variables:

Functions should take parameters instead of relying on global variables. This makes the function more reusable and testable

Good Example:


python

 Copy code

```
def calculate_discount(price, discount_percentage):  
    return price - (price * (discount_percentage / 100))
```

Bad Example:

python

 Copy code

```
discount_percentage = 10 # Global variable  
  
def calculate_discount(price):  
    return price - (price * (discount_percentage / 100))
```

*ARGS and **KWARGS:

*args


Purpose: Allows you to pass a variable number of positional arguments to a function.

How It Works: When you add `*args` in a function, it collects all the additional positional arguments passed to the function and stores them in a tuple.

Usage: This is helpful when you don't know beforehand how many arguments will be passed to the function.

Example:

python

 Copy code

```
def multiply(*args):
    result = 1
    for num in args:
        result *= num
    return result

print(multiply(2, 3, 4)) # Output: 24
```

**kwargs

Purpose: Allows you to pass a variable number of keyword arguments to a function.

How It Works: When you add `**kwargs` in a function, it collects all the additional keyword arguments passed to the function and stores them in a dictionary.

Usage: This is useful for functions that might need to handle various optional settings.

python

 Copy code

```
def greet(**kwargs):
    if 'name' in kwargs:
        print(f"Hello, {kwargs['name']}!")
    else:
        print("Hello, Guest!")

greet(name="Alice") # Output: Hello, Alice!
greet(age=30)      # Output: Hello, Guest!
```

Using `*args` and `**kwargs` Together


Key Principles

Positional and Keyword Flexibility: `*args` allows you to accept multiple positional arguments, and `**kwargs` allows multiple keyword arguments.

Order Matters: When defining a function, `*args` must come before `**kwargs`.

Helpful for Dynamic Functions: These are especially useful in functions that need to accept various configurations or in wrapper functions where arguments need to be passed dynamically.

python

 Copy code

```
def example_function(*args, **kwargs):  
    print("Positional arguments:", args)  
    print("Keyword arguments:", kwargs)  
  
example_function(1, 2, 3, name="Alice", age=30)  
# Output:  
# Positional arguments: (1, 2, 3)  
# Keyword arguments: {'name': 'Alice', 'age': 30}
```

LAMBDA FUNCTIONS:

Syntax:

python

 Copy code

```
lambda arguments: expression
```

A lambda function can take any number of arguments but only a single expression.

The expression is evaluated and returned. Lambda functions don't use a return statement; the result of the expression is automatically returned.

Anonymous Functions:

Lambda functions are unnamed (anonymous) functions, typically used for one-off operations or simple tasks that don't need a separate function name.

Single Expression:

They're limited to a single expression, making them ideal for small calculations or quick transformations, not complex logic.


Usage in Higher-Order Functions:

Lambda functions are often used with functions that take other functions as arguments, such as `map()`, `filter()`, and `sorted()`.

Examples

1. Basic Lambda Function:

python

 Copy code

```
add = lambda x, y: x + y
print(add(5, 3)) # Output: 8
```

2. Using Lambda with `map()`:


python

 Copy code

```
numbers = [1, 2, 3, 4]
squares = list(map(lambda x: x**2, numbers))
print(squares) # Output: [1, 4, 9, 16]
```

3. Using Lambda with `sorted()`:

python

 Copy code

```
names = ["Alice", "Bob", "Cathy"]
sorted_names = sorted(names, key=lambda x: len(x))
print(sorted_names) # Output: ['Bob', 'Alice', 'Cathy']
```

OOP Basics: Encapsulation


Definition: Encapsulation is the concept of bundling the data (attributes) and methods (functions) that operate on the data into a single unit, or class. It also restricts access to certain components, protecting the internal state of an object from unintended interference.

Purpose: To hide an object's internal state and only expose necessary functionalities, ensuring controlled access to the data.

Example: Using private or protected attributes (`_attribute` or `__attribute`) and providing getter and setter methods to control access.

Example:

python

 Copy code

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

    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        if amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient balance.")

    def get_balance(self):
        return self.__balance

account = BankAccount(100)
account.deposit(50)
print(account.get_balance()) # Output: 150
```


Abstraction


Definition: Abstraction focuses on hiding the complex implementation details of a system and exposing only the essential features. It defines methods that need to be implemented but doesn't specify the details.

Purpose: To simplify the code interface, allowing users to work at a higher level without needing to understand complex underlying logic.

Example: Using abstract classes and methods, often with `@abstractmethod` in Python, which enforce a contract for subclasses.

Example:

python

 Copy code

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        return "Woof!"

class Cat(Animal):
    def sound(self):
        return "Meow!"

# Usage
dog = Dog()
print(dog.sound()) # Output: Woof!
```

Inheritance


Definition: Inheritance allows a new class (subclass) to inherit properties and behaviors (methods and attributes) from an existing class (superclass or parent class). This creates a hierarchical relationship between classes.

Purpose: To promote code reuse, allowing new classes to use and extend existing functionalities without rewriting code.

Example: Inheriting from a base class and adding or overriding methods.

Example:

python

 Copy code

```
class Vehicle:
    def start(self):
        print("Vehicle started.")

class Car(Vehicle):
    def honk(self):
        print("Car honking.")

# Usage
car = Car()
car.start() # Output: Vehicle started.
car.honk()  # Output: Car honking.
```

Polymorphism


Definition: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables a single function or method to work in different ways depending on the object calling it.

Purpose: To allow for flexibility in code, where different objects can be treated similarly, especially when using interfaces or abstract classes.

Example: Method overriding and using the same interface or method name in multiple classes.

Example:

python

 Copy code

```
class Bird:
    def fly(self):
        print("Bird is flying.")

class Airplane:
    def fly(self):
        print("Airplane is flying.")

# Polymorphic function
def make_it_fly(obj):
    obj.fly()

# Usage
bird = Bird()
plane = Airplane()

make_it_fly(bird)    # Output: Bird is flying.
make_it_fly(plane)  # Output: Airplane is flying.
```


What to Put in the `__init__` Function:

Core attributes of the object: These are the key characteristics that define your object. For example, for a `Circle` class, the **radius** is a core attribute because it directly defines the shape of the circle.

Parameters for required setup: These are the pieces of information needed to create a valid object. For example, if your class is a `Car`, the `__init__` function might require parameters like `make`, `model`, and `year` to properly initialize a `Car` object.

Attributes that can't be derived easily: If you need some information that can't be easily computed from other attributes, it should be included in the `__init__` function. For example, if a class represents a person, you might store both `first_name` and `last_name` explicitly because both are needed individually, not just derived from one or the other.

python

 Copy code

```
class Car:
    def __init__(self, make, model, year, color):
        self.make = make        # Store the make of the car
        self.model = model      # Store the model of the car
        self.year = year        # Store the year of the car
        self.color = color      # Store the color of the car
```

Decorator:

A **decorator** is a function that takes another function as an argument, adds some functionality to it, and returns a new function.

Decorators are denoted with the `@decorator_name` syntax above a function to apply them.

Basic Decorator Structure:

A **decorator** function typically wraps the input function in an inner function, which then calls the input function along with any additional logic.

Example:

```
python Copy code

def my_decorator(func):
    def wrapper():
        print("Something before the function.")
        func()
        print("Something after the function.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

Output:

```
bash Copy code

Something before the function.
Hello!
Something after the function.
```

Using `*args` and `**kwargs` for Flexibility

Decorators often use `*args` and `**kwargs` to handle functions with any number of arguments.

Example:

python

 Copy code

```
def my_decorator(func):  
    def wrapper(*args, **kwargs):  
        print("Before function")  
        result = func(*args, **kwargs)  
        print("After function")  
        return result  
    return wrapper
```

Common Use Cases for Decorators

- **Logging:** Track function calls and outputs.
- **Access Control:** Restrict function access based on conditions.
- **Timing:** Measure function execution time.
- **Memoization:** Cache results of expensive functions.

Built-in Decorators

Python provides built-in decorators like:

@staticmethod and **@classmethod**: Define methods within classes that aren't bound to instance or class variables.

@property: Allows a method to be accessed like an attribute.

@staticmethod

Purpose: Indicates that a method does not require access to the instance (`self`) or the class (`cls`). It behaves like a regular function but resides in the class's namespace for organizational purposes.

Use case: When a method performs a task that relates to the class but does not depend on any instance or class variables.

Example:

```
python Copy code  
  
class MathOperations:  
    @staticmethod  
    def add(a, b):  
        return a + b  
  
# Usage  
print(MathOperations.add(3, 5)) # Outputs: 8
```

@classmethod

Purpose: Similar to `@staticmethod`, but instead of taking a `self` parameter (instance), it takes `cls` as its first parameter, which represents the class itself. This allows `@classmethod` to access and modify class-level attributes.

Use case: When a method needs to operate on the class itself rather than instances, or when it's meant to create or return an instance of the class (often used as an alternative constructor)

Example:

```
python Copy code  
  
class Dog:  
    species = "Canis lupus familiaris" # Class attribute  
  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    @classmethod  
    def create_puppy(cls, name):  
        return cls(name, 0) # Returns a new Dog instance with age 0  
  
# Usage  
puppy = Dog.create_puppy("Buddy")  
print(puppy.name, puppy.age) # Outputs: Buddy 0
```

@property

Purpose: Allows a method to be accessed like an attribute rather than calling it with parentheses (). This is useful when you want to control access to an attribute or compute it dynamically, while keeping a clean interface.

Use case: When you want to add logic to an attribute access, like performing calculations or validations.

Example:

```
python Copy code  
  
class Rectangle:  
    def __init__(self, width, height):  
        self.width = width  
        self.height = height  
  
    @property  
    def area(self):  
        return self.width * self.height # No need to call this as a method  
  
# Usage  
rect = Rectangle(4, 5)  
print(rect.area) # Outputs: 20
```

Why These Built-in Decorators are Important

Code Organization: They help organize functions by keeping related utility methods within classes (@staticmethod), making alternative constructors (@classmethod), and creating controlled, read-only, or computed properties (@property).

Encapsulation: @property enables data encapsulation, allowing computed values to be accessed as attributes, while also giving you control over attribute access.

Simplicity and Readability: These decorators improve readability by allowing logical methods to be accessed more intuitively and keeping them tied to the relevant classes.

API (Application Programming Interface): Making HTTP Requests

The Python `requests` library is commonly used to interact with web APIs.

HTTP Methods: Use appropriate HTTP methods (GET, POST, PUT, DELETE) to interact with the API.

- **GET:** Used to retrieve data.
- **POST:** Used to send data to create a new resource.
- **PUT:** Used to update an existing resource.
- **DELETE:** Used to delete a resource.

Example:

```
python

import requests

response = requests.get("https://api.example.com/data")
data = response.json()  # Converts JSON response to a Python dictionary
```

 Copy code

Handling Responses

Status Codes: Check the HTTP status code to confirm if the request was successful.

- **200-299:** Success codes (e.g., `200 OK`, `201 Created`).
- **400-499:** Client errors (e.g., `404 Not Found`).
- **500-599:** Server errors (e.g., `500 Internal Server Error`).

JSON Responses: Most APIs return data in JSON format, which `requests` can parse with `.json()`.

Example:

python

 Copy code

```
if response.status_code == 200:
    data = response.json()
    print(data)
else:
    print("Error:", response.status_code)
```

Access Modes	Description
r	Opens a file for reading only.
rb	Opens a file for reading only in binary format.
r+	Opens a file for both reading and writing.
rb+	Opens a file for both reading and writing in binary format.
w	Opens a file for writing only.
wb	Opens a file for writing only in binary format.
w+	Opens a file for both writing and reading.
wb+	Opens a file for both writing and reading in binary format.
a	Opens a file for appending.
ab	Opens a file for appending in binary format.
a+	Opens a file for both appending and reading.
ab+	Opens a file for both appending and reading in binary format.