Constructors and `__init__()` in Python

Introduction

In the previous lesson, we learned that a **class** is a blueprint and an **object** is an instance created from that class.

Now an important question comes up:

> When an object is created, how does it get its initial data?

For example, if we create:

```
student = Student("Hafsa", 25, 9)
```

How does Python know that:

- `"Hafsa"` should become the name
- `25` should become the age
- `9` should become the grade?

This is where `__init__()` becomes important.

`__init__()` is a special method that runs automatically when a new object is initialized.

---

1. What Is a Constructor?

A **constructor** is the mechanism used to initialize a newly created object with its initial state.

In Python, beginners often hear:

> "__init__() is the constructor."

This is a useful simplification, but technically Python separates **object creation** from **object initialization**.

The important practical idea is:

```
Create object
    ↓
Initialize object
    ↓
Object is ready to use
```

In normal Python classes, __init__() is the method you use to initialize the object's attributes.

---

2. What Is __init__()?

__init__() is a special method defined inside a class.

Example:

```
class Student:

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

When we create:

```
student = Student("Hafsa", 25)
```

Python automatically calls:

```
__init__(student, "Hafsa", 25)
```

Conceptually, the object is initialized with:

```
student
□□□ name □ "Hafsa"
□□□ age  □ 25
```

You normally don't call `__init__()` yourself.

---

3. Why Do We Need `__init__()`?

Without `__init__()`, we could create an object and then assign attributes manually:

```
class Student:
    pass

student = Student()

student.name = "Hafsa"
student.age = 25
student.grade = 9
```

This works, but it becomes repetitive.

With `__init__()`:

```
class Student:

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

We can simply write:

```
student = Student("Hafsa", 25, 9)
```

The object is initialized immediately.

---

4. Basic Structure

The common pattern is:

```
class ClassName:

    def __init__(self, parameters):
        self.attribute = value
```

Example:

```
class Car:

    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
```

Create an object:

```
car = Car("Toyota", "Corolla")
```

Now:

```
print(car.brand)
print(car.model)
```

Output:

```
Toyota
Corolla
```

---

5. Understanding the Parameters

Look at:

```
class Student:

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

There are three important parts:

```
self
name
age
```

self

Represents the current object.

name

A parameter receiving the value passed during object creation.

age

Another parameter receiving the value passed during object creation.

When we write:

```
student = Student("Hafsa", 25)
```

the values are passed into __init__().

Conceptually:

```
self  student
name  "Hafsa"
age   25
```

Then:

```
self.name = name
```

means:

```
student.name = "Hafsa"
```

And:

```
self.age = age
```

means:

```
student.age = 25
```

---

6. `self.name = name`

This line is extremely important:

```
self.name = name
```

It may look confusing because `name` appears twice.

But they represent different things.

`self.name`

Attribute belonging to the object

`name`

Parameter containing the incoming value

For example:

```
student = Student("Hafsa", 25)
```

The `"Hafsa"` value enters the `name` parameter.

Then:

```
self.name = name
```

stores that value inside the object.

So eventually:

`student.name  "Hafsa"`

---

7. `__init__()` Runs Automatically

Consider:

```
class User:

    def __init__(self, username):
        print("Initializing user...")
        self.username = username
```

Now:

```
user = User("hafsa")
```

Output:

```
Initializing user...
```

We didn't write:

```
user.__init__()
```

Python automatically called it during initialization.

---

8. Creating Multiple Objects

Every time you create an object, `__init__()` runs for that object.

```
class Student:

    def __init__(self, name, grade):
        self.name = name
        self.grade = grade
```

Create three students:

```
student1 = Student("Hafsa", 9)
student2 = Student("Asim", 10)
student3 = Student("Ali", 9)
```

The initialization process happens separately for each object.

```
student1
name = Hafsa
grade = 9

student2
name = Asim
grade = 10

student3
name = Ali
grade = 9
```

Each object gets its own instance attributes.

---

9. Default Values in `__init__()`

Parameters can have default values.

```
class User:

    def __init__(self, name, role="user"):
        self.name = name
        self.role = role
```

Now:

```
user1 = User("Hafsa")
user2 = User("Asim", "admin")
```

The values become:

```
user1
name = Hafsa
role = user

user2
name = Asim
role = admin
```

This is useful when most objects should have the same default behavior.

---

10. Required Arguments

If a parameter has no default value, it is required.

```
class Product:
```

```
    def __init__(self, name, price):
        self.name = name
        self.price = price
```

This works:

```
product = Product("Keyboard", 50)
```

But this does not:

```
product = Product("Keyboard")
```

Python raises a `TypeError` because `price` is missing.

This helps ensure that objects are created with the information they need.

---

11. Initializing Different Data Types

`__init__()` can initialize any type of data.

```
class Profile:

    def __init__(self, name, age, is_active, skills):
        self.name = name
        self.age = age
        self.is_active = is_active
        self.skills = skills
```

Create an object:

```
profile = Profile(
```

```
    "Hafsa",
    25,
    True,
    ["Python", "JavaScript", "Git"]
)
```

Now the object contains:

```
name      → string
age       → integer
is_active → boolean
skills    → list
```

---

12. Initializing an Empty Collection

A common pattern is to initialize an empty list inside `__init__()`.

```
class ShoppingCart:

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

Create two carts:

```
cart1 = ShoppingCart()
cart2 = ShoppingCart()
```

Each object gets its own list.

```
cart1.items.append("Keyboard")
```

Now:

```
print(cart1.items)
print(cart2.items)
```

Output:

```
['Keyboard']
[]
```

This is important because the two carts should not share the same item list.

---

13. A Common Mistake With Mutable Defaults

Avoid this pattern:

```
class ShoppingCart:

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

Using a mutable object such as a list as a default argument can cause unexpected shared state.

Prefer:

```
class ShoppingCart:

    def __init__(self, items=None):
        self.items = [] if items is None else items
```

Now each cart gets a fresh empty list when no list is supplied.

---

14. Calling Other Methods From `__init__()`

An initializer can call another instance method.

```
class User:

    def __init__(self, name):
        self.name = name
        self.setup()

    def setup(self):
        print(f"Setting up {self.name}")
```

Create:

```
user = User("Hafsa")
```

Output:

```
Setting up Hafsa
```

This can be useful, but keep initialization simple and predictable.

---

15. Validation Inside `__init__()`

One useful purpose of initialization is validating incoming data.

```
class Student:

    def __init__(self, name, age):
        if age < 0:
```

```
            raise ValueError("Age cannot be negative")

        self.name = name
        self.age = age
```

Now:

```
student = Student("Hafsa", -5)
```

raises an error instead of creating an invalid object.

The general idea is:

```
Input
  
Validate
  
Store valid state
  
Object ready
```

---

16. Keeping Objects in a Valid State

Good classes try to prevent invalid states.

For example:

```
class BankAccount:

    def __init__(self, owner, balance=0):
        if balance < 0:
```

```
            raise ValueError("Balance cannot be negative")

        self.owner = owner
        self.balance = balance
```

Now:

```
account = BankAccount("Hafsa", 5000)
```

is valid.

But:

```
account = BankAccount("Hafsa", -100)
```

raises an error.

This becomes increasingly important as applications become larger.

---

17. `__init__()` Does Not Return the Object

A common misconception is that `__init__()` creates and returns the object.

It doesn't.

`__init__()` initializes an object that has already been created.

Therefore, you should not return another value from it.

Incorrect:

```
class Student:

    def __init__(self, name):
        self.name = name
        return self
```

__init__() must return None.

Normally you simply write:

```
class Student:

    def __init__(self, name):
        self.name = name
```

---

18. Object Creation vs Object Initialization

This distinction is useful for understanding Python more deeply.

When you write:

```
student = Student("Hafsa")
```

Python needs to:

```
1. Create an object
        
2. Initialize that object
        
3. Assign it to student
```

Python's object creation mechanism involves `__new__()`.

Object initialization is handled by:

`__init__()`

For normal Python development, you will usually work with `__init__()` and rarely need to override `__new__()`.

So remember:

`__new__()`

creates the instance

`__init__()`

initializes the instance

---

19. Why You Usually Don't Write `__new__()`

For most classes, Python handles object creation automatically.

You simply write:

```
class User:

    def __init__(self, name):
        self.name = name
```

Then:

`user = User("Hafsa")`

That's enough.

`__new__()` becomes relevant for advanced cases such as:

- immutable types
- customized object creation
- certain singleton patterns
- metaprogramming

Those are topics for advanced Python.

For everyday classes:

> Focus on `__init__()`.

---

20. A Real Example: Resource

Suppose haas.dev needs a `Resource` object.

A resource may require:

- title
- description
- category
- file name

We can initialize all of this:

```
class Resource:

    def __init__(self, title, description, category, file_name):
        self.title = title
        self.description = description
        self.category = category
        self.file_name = file_name
```

Create a resource:

```
resource = Resource(
    "Python Functions",
    "Learn how Python functions work.",
    "Python",
    "python-functions.pdf"
)
```

Now:

```
print(resource.title)
print(resource.category)
print(resource.file_name)
```

Output:

```
Python Functions
Python
python-functions.pdf
```

The object starts life with all the information it needs.

---

21. `__init__()` With More Complex Initialization

A class can initialize calculated values too.

```
class Product:

    def __init__(self, name, price, quantity):
        self.name = name
        self.price = price
        self.quantity = quantity
        self.total = price * quantity
```

Create:

```
product = Product("Keyboard", 50, 3)
```

Now:

```
print(product.total)
```

Output:

```
150
```

The `total` value was calculated during initialization.

However, if `price` or `quantity` can change later, it may be better to calculate the total when needed rather than store a potentially outdated value.

This is an important design consideration.

---

22. Initialization Should Have a Clear Responsibility

A good `__init__()` usually answers:

> What does this object need to exist in a valid initial state?

For example:

```
class User:

    def __init__(self, username, email):
        self.username = username
        self.email = email
```

Good.

But putting too much unrelated work inside `__init__()` can make a class difficult to use.

Avoid turning initialization into a huge process such as:

```
Create object
 
Connect to five services
 
Download files
 
Process thousands of records
 
Send emails
 
```

`Start background tasks`

Initialization should generally be focused on establishing the object's initial state.

---

23. Constructor Patterns

Different classes need different initialization strategies.

No required data

```
class Counter:

    def __init__(self):
        self.count = 0
```

Required data

```
class User:

    def __init__(self, name, email):
        self.name = name
        self.email = email
```

Required + optional data

```
class User:

    def __init__(self, name, role="user"):
        self.name = name
        self.role = role
```

Validation

```
class Product:

    def __init__(self, price):
        if price < 0:
            raise ValueError("Price cannot be negative")

        self.price = price
```

Choose the pattern according to what the object actually needs.

---

24. Constructor Chaining With Inheritance

When inheritance is introduced, a child class may need to initialize data from its parent.

Example:

```
class User:

    def __init__(self, name):
        self.name = name


class Admin(User):

    def __init__(self, name, permissions):
        super().__init__(name)
        self.permissions = permissions
```

Create:

```
admin = Admin("Hafsa", ["edit", "delete"])
```

Now the object has:

`name`
`permissions`

`super().__init__(name)` calls the parent's initializer.

This becomes especially useful when working with inheritance.

---

25. Common Beginner Mistakes

Mistake 1: Misspelling `__init__`

Correct:

```
def __init__(self):
    ...
```

The name must contain:

- two underscores before `init`
- two underscores after `init`

`__init__`

Not:

`_init`
`init`
`__init`
`init__`

Mistake 2: Forgetting `self`

Incorrect:

```
class User:

    def __init__(name):
        self.name = name
```

Correct:

```
class User:

    def __init__(self, name):
        self.name = name
```

Mistake 3: Forgetting to store the parameter

This:

```
class User:

    def __init__(self, name):
        name = name
```

doesn't create an instance attribute.

Correct:

```
class User:
```

```
    def __init__(self, name):
        self.name = name
```

Mistake 4: Passing the wrong number of arguments

Given:

```
class User:

    def __init__(self, name, email):
        self.name = name
        self.email = email
```

This is incorrect:

```
user = User("Hafsa")
```

because `email` is required.

Mistake 5: Returning a value from `__init__()`

Don't do:

```
def __init__(self):
    return "hello"
```

`__init__()` must return `None`.

Mistake 6: Putting too much logic into initialization

Don't make `__init__()` responsible for the entire application.

Its main responsibility is to establish the object's initial state.

---

26. Mental Model

When you see:

```
student = Student("Hafsa", 25)
```

think:

```
Student
  
Python creates an instance
  
__init__() receives the values
  
self refers to that instance
  
attributes are initialized
  
student is ready to use
```

For example:

```
class Student:

    def __init__(self, name, age):
```

```
        self.name = name
        self.age = age
```

Think:

```
Student("Hafsa", 25)
        □
        □□□ name □ "Hafsa"
        □□□ age  □ 25
                 □
            self.student
```

The exact internal mechanics are more detailed, but this mental model is enough for normal development.

---

27. Practical Design Checklist

Before writing `__init__()`, ask:

What does this object need?

```
User
□ name
□ email
```

Which values are required?

```
name □ required
email □ required
```

Which values can have defaults?

```
role □ "user"
```

What should be validated?

`age >= 0`
`price >= 0`
`email format`

What should be initialized?

`instance attributes`
`empty collections`
`initial state`

What should NOT happen here?

Avoid unnecessary:

- heavy processing
- unrelated side effects
- complicated application logic

This makes classes easier to understand and maintain.

---

28. Mini Project: User Account

Create a `User` class with:

`username`
`email`
`role`
`is_active`

Use:

```
class User:

    def __init__(self, username, email, role="user"):
        self.username = username
        self.email = email
        self.role = role
        self.is_active = True
```

Create:

```
user1 = User("hafsa", "hafsa@example.com")
user2 = User("asim", "asim@example.com", "admin")
```

Display:

```
print(user1.username)
print(user1.role)
print(user1.is_active)

print(user2.username)
print(user2.role)
print(user2.is_active)
```

Extend the project

Add methods:

```
deactivate()
activate()
display_profile()
```

Then create at least three users and test their behavior.

---

29. 5 Minute Challenge

Create a `BankAccount` class.

The initializer should receive:

`owner`
`balance`

The balance should default to `0`.

Example:

```
account = BankAccount("Hafsa")
```

should create:

```
owner = Hafsa
balance = 0
```

Also make sure a negative starting balance raises an error.

Bonus: Add:

`deposit()`
`withdraw()`

methods.

---

30. Exercises

Exercise 1

Create a `Book` class with:

`title`
`author`
`pages`

Initialize all three values through `__init__()`.

Exercise 2

Create a `Rectangle` class with:

`width`
`height`

Give `width` and `height` default values of `1`.

Exercise 3

Create a `Product` class with:

`name`
`price`
`stock`

Validate that price and stock cannot be negative.

Exercise 4

Create a `Task` class with:

`title`
`completed`

Make `completed` default to `False`.

Exercise 5

Create a `Course` class with:

`name`
`instructor`
`students`

Initialize `students` as an empty list.

Make sure each course receives its own separate list.

---

31. Mini Quiz

Question 1

What is the main purpose of `__init__()`?

A. Delete an object
B. Initialize an object's state

C. Import a module
D. Stop a loop

Answer: B

Question 2

When does `__init__()` normally run?

A. When Python starts
B. When a new instance is initialized
C. Only when manually called
D. When a module is imported

Answer: B

Question 3

What does `self` refer to inside `__init__()`?

A. The class
B. The current object
C. The Python interpreter
D. The parent class

Answer: B

Question 4

What is the difference between `name` and `self.name`?

Answer:

`name` is a parameter.

`self.name` is an attribute stored on the current object.

Question 5

Which is correct?

```
class User:

    def __init__(self, name):
        self.name = name
```

or:

```
class User:

    def init(self, name):
        self.name = name
```

Answer: The first one.

`__init__()` must have the exact special-method name.

---

32. Interview Questions

Beginner

1. What is `__init__()` in Python?
2. Why do we use `__init__()`?
3. Does `__init__()` run automatically?
4. What does `self` represent?
5. What is an instance attribute?
6. Can `__init__()` have parameters?
7. Can parameters have default values?
8. What happens if a required constructor argument is missing?

Intermediate

9. Is `__init__()` technically responsible for creating the object?
10. What is the difference between `__new__()` and `__init__()`?
11. Why should `__init__()` return `None`?
12. Why can mutable default arguments be dangerous?
13. How can you validate data inside `__init__()`?
14. Why should initialization logic be kept focused?
15. How does `super().__init__()` work with inheritance?

---

33. Cheat Sheet

Basic initializer

```
class User:

    def __init__(self, name):
        self.name = name
```

Create object

```
user = User("Hafsa")
```

Multiple parameters

```
class Product:

    def __init__(self, name, price):
        self.name = name
        self.price = price
```

Default value

```
class User:

    def __init__(self, name, role="user"):
        self.name = name
        self.role = role
```

Validation

```
class Product:

    def __init__(self, price):
        if price < 0:
            raise ValueError("Invalid price")

        self.price = price
```

Empty list

```
class Cart:

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

Parent initialization

```
class Admin(User):

    def __init__(self, name, permissions):
        super().__init__(name)
        self.permissions = permissions
```

Object creation vs initialization

```
__new__()
    ↓
Creates instance

__init__()
    ↓
Initializes instance
```

---

34. Key Takeaways

- `__init__()` is a special Python method used to initialize an object.
- It normally runs automatically when an instance is created.
- `self` represents the current instance.
- Parameters receive values passed during object creation.
- `self.attribute = value` stores those values in the object.
- `__init__()` can provide default values.
- It can validate data before storing it.
- Each object receives its own instance attributes.
- Mutable attributes such as lists should generally be created inside `__init__()`.
- `__init__()` initializes an object; technically, object creation is handled separately through `__new__()`.
- You normally don't need to override `__new__()`.

- A good initializer establishes a clear and valid initial state.
- `super().__init__()` can be used to initialize inherited parent-class state.

The core pattern to remember is:

```
Class
  
__init__()
  
Initial data
  
Instance attributes
  
Ready-to-use object
```

Website: https://dev-roast-app.vercel.app

Visit haas.dev for more resources and guides.