Python does NOT reserve the word self.
You can technically write:

```
def f(this, x):
    print(this.x)
```

self represents the current object calling the method

Example with multiple objects:

```
class Person:
    def set_name(self, name):
        self.name = name

p1 = Person()
p2 = Person()

p1.set_name("Alice")
p2.set_name("Bob")
```

Person.set_name(p1, "Alice")
Person.set_name(p2, "Bob")So:

self = p1 in first call

self = p2 in second call
This is how Python keeps object data separate

Why MUST we pass self in class methods?Because Python does NOT automatically know:
which object's data to use
which object's variables to modify

When you call:
obj.f()
Python automatically does this:
A.f(obj)
This happens before your function body executes

Your code

```
class A:
        def f():
        print(x)

obj = A()
obj.f()
```

What Python actually tries to do
A.f(obj)
But your function definition is:

```
def f():
```

That means:
Function expects 0 arguments
Python passes 1 argument (obj)
CRASH happens here
Actual error (important):
TypeError: f() takes 0 positional arguments but 1 was given

🟢 Case 2 — Method does NOT use instance data

```python
class A:
    def f():
        print("Hello")
```

This works ONLY when called like:

```python
A.f()
```

Because:

```python
A.f()
```

passes **no object**, so signature matches.

But this fails:

```python
obj.f()
```

Because Python again does:

```python
A.f(obj)
```

💥 Same error: extra argument

🟢 Your rewritten example (let's analyze it correctly)

You wrote:

```python
class A:
    def f(self, x):
        print(self.x)

obj = A()
obj.f(10)
```

What happens here?

Python does:

```python
A.f(obj, 10)
```

✔ Signature matches (`self, x`)
❌ But you print `self.x`, NOT `x`

Result:

```pgsql
AttributeError: 'A' object has no attribute 'x'
```

Because you never did:

```python
self.x = x
```

✅ Correct version

```python
class A:
    def f(self, x):
        self.x = x
        print(self.x)

obj = A()
obj.f(10)
```

✔ Works perfectly

🟢 About “x is a class variable”

❌ **This is incorrect**

```python
x
```

by itself is:

- NOT a class variable
- NOT an instance variable
- Just a **name lookup**

To be a **class variable**:

```python
class A:
    x = 10
```

To be an **instance variable**:

```python
self.x = 10
```

When calling a method:

Call style	Python converts it to
`obj.f()`	`A.f(obj)`
`obj.f(10)`	`A.f(obj, 10)`
`A.f()`	`A.f()`

****

Instance variable	One specific object	`self.x = ...`
Class variable	The class itself (shared)	`ClassName.x = ...`

2️⃣ Instance Variable (MOST COMMON)

Example

```python
class Person:
    def __init__(self, name):
        self.name = name   # instance variable
```

Usage

```python
p1 = Person("Alice")
p2 = Person("Bob")

print(p1.name)  # Alice
print(p2.name)  # Bob
```

Why?

Each object has its **own copy.**

Mental model

🧠 *"Instance variables live inside the object."*

Changing one does NOT affect others

```python
p1.name = "Charlie"

print(p1.name)  # Charlie
print(p2.name)  # Bob
```

3️⃣ Class Variable (SHARED by ALL objects)

Example

```python
class Person:
    species = "Human"   # class variable
```

Usage

```python
p1 = Person()
p2 = Person()

print(p1.species)
print(p2.species)
```

Output:

```nginx
Human
Human
```

Mental model

🧠 *"Class variables live in the blueprint, not the object."*

Changing class variable

```python
Person.species = "Alien"

print(p1.species)
print(p2.species)
```

Output:

```nginx
Alien
```

4️⃣ THE MOST IMPORTANT RULE (THIS CAUSES CONFUSION)

❌ THIS creates an INSTANCE variable, not a class variable

```python
p1.species = "Robot"
```

Now:

```python
print(p1.species)  # Robot
print(p2.species)  # Alien
```

Why?

Python says:

> "If the object has its own variable, use that FIRST."

This is called **shadowing**.

So here you have assigned a value for the object p1 (p1.species="Robot") so thats why it is an Instance Variable here.

If you change the value through a class name like Person.species = "Alien", Then it is a CLASS Variable.

When Python sees:

`obj.x`

It checks in this order:

1. Instance (`obj.__dict__`)
2. Class (`Class.__dict__`)
3. Parent classes

we usually define instance variables in `__init__`

```
class A:

    def __init__(self, x):

        self.x = x
```

✓ Guarantees object always has `x`
✓ Prevents runtime errors
✓ Professional practice

9️⃣ Quick comparison table (SAVE THIS)

Feature	Instance Variable	Class Variable
Defined using	`self.x`	`Class.x`
Unique per object	✅ Yes	❌ No
Shared	❌ No	✅ Yes
Stored in	object memory	class memory
Common use	object state	constants, counters

Q3 — ❌ Incorrect (this reveals class-variable confusion)

Code

```python
class C:
    total = 0

    def __init__(self):
        C.total += 1

c1 = C()
c2 = C()

print(C.total)
```

Your answer

> `1`

✅ Correct answer

```
2
```

Why?

- `C.total` starts at `0`
- `c1 = C()` → `total = 1`
- `c2 = C()` → `total = 2`

`total` is a **class variable**, shared and incremented every time an object is created.
