Object-oriented
Methods
in **Python** !

Introducing: Methods

- A **method** is a special kind of function defined in a class.
  - The first parameter, idiomatically named **self**, is special (coming next!)
  - Everything else you know about a function's parameters, return types, and evaluation rules are the same with methods.

```
class ClassName:

    ... # Attributes Elided

    def method_name(self, [params...]) -> retT:
        <method body>
```

- Once defined, you can call a method ***on*** any object of that class using the dot operator.
  - Just like how attributes were accessed except followed by parenthesis and any necessary arguments *excluding one for self.*

```
an_object: ClassName = ClassName()
an_object.method_name()
```

Functions vs. *Methods*

1. Let's define a *silly* **function.**

```
def say_hello() -> None:
    print("Hello, world")
```

2. Once defined, we can then call it.

```
say_hello()
```

3. Now, let's define that same function as a **method** *of the* `Person` *class.*

```
class Person:

    ...   # attributes elided

    def say_hello(self) -> None:
        print("Hello, world.")
```

4. Once defined, we can call the method on any `Person` object:

```
a_person: Person = Person()
a_person.say_hello()
```

Hands-on: Practice with the `self` parameter

1. Declare a **name** attribute of type **str**

2. Initialize the name attribute of the Person object you construct in the main function

3. Update the say_hello method as shown to the right. *Notice the conversion to an f-string!*

4. Try constructing *another* person object in main and also calling its say_hello method.

```python
def say_hello(self) -> None:
    print(f"Hello, I'm {self.name}!")
```

A Method's Superpower is that it automagically gets a *reference* to the object the method was called on!

- Consider the method call:

```
a_person.say_hello()
```

  - The object reference is `a_person`
  - The method being called is `say_hello()`

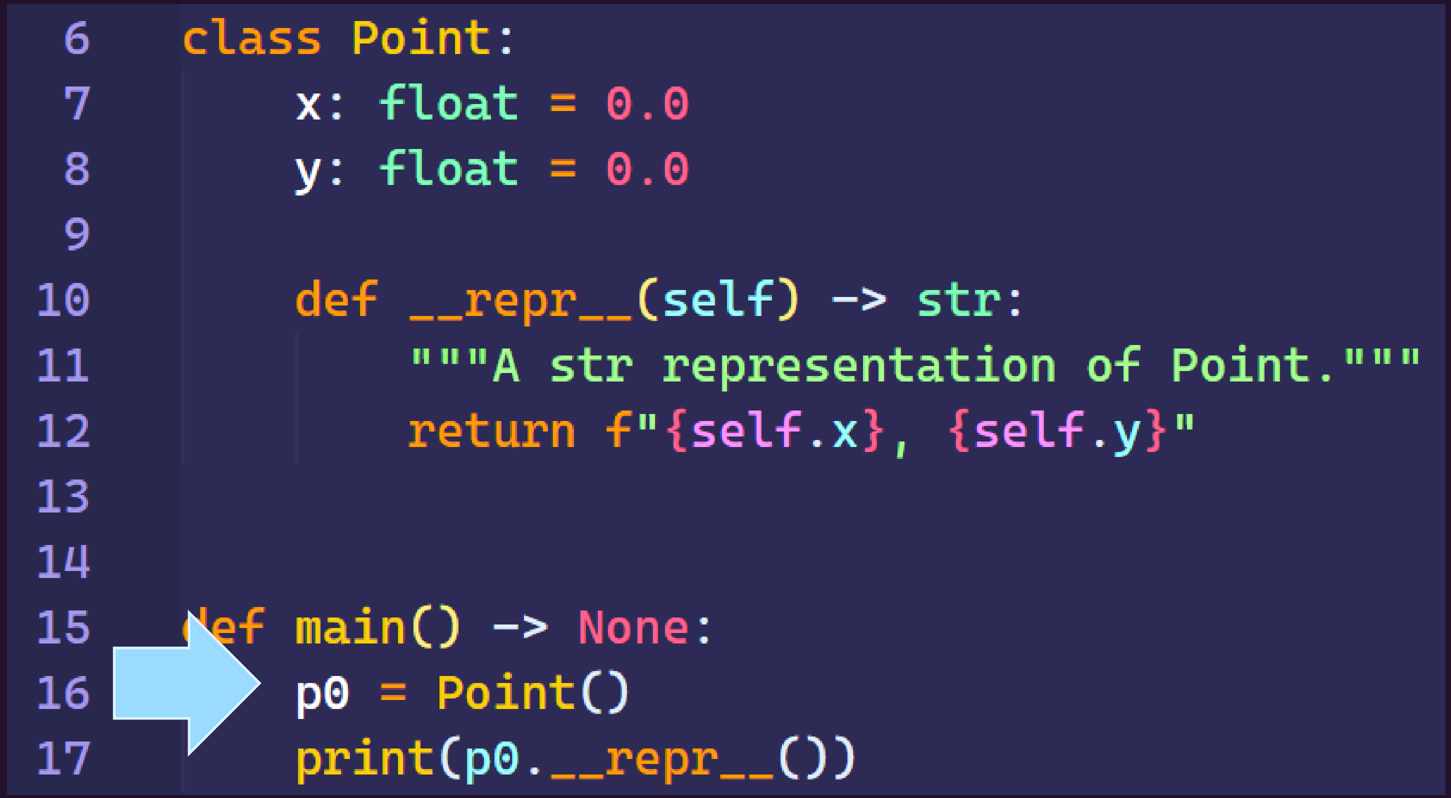
- The say_hello method's definition is:

```
class Person:
    ...  # Attributes Elided
    def say_hello(self) -> None:
        print(f"Hello, I'm {self.name}!")
```

- Notice: The method has an untyped first parameter named `self`.
  - Its type is *implicitly* the same as the class it is defined in.

- When a method call evaluates, the object reference is automagically its first argument.
  - Thus, in the example above, `self` would refer to the same object that `a_person` does.

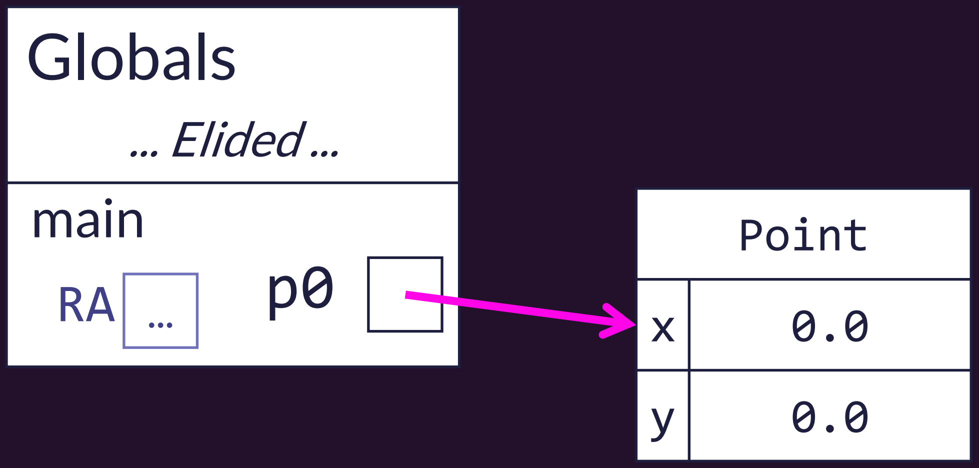
Suppose the interpreter *just* completed this line...



```
class Point:
    x: float = 0.0
    y: float = 0.0

    def __repr__(self) -> str:
        """A str representation of Point."""
        return f"{self.x}, {self.y}"


def main() -> None:
    p0 = Point()
    print(p0.__repr__())
```

The Stack

The Heap



Globals

... Elided ...

main

RA ...

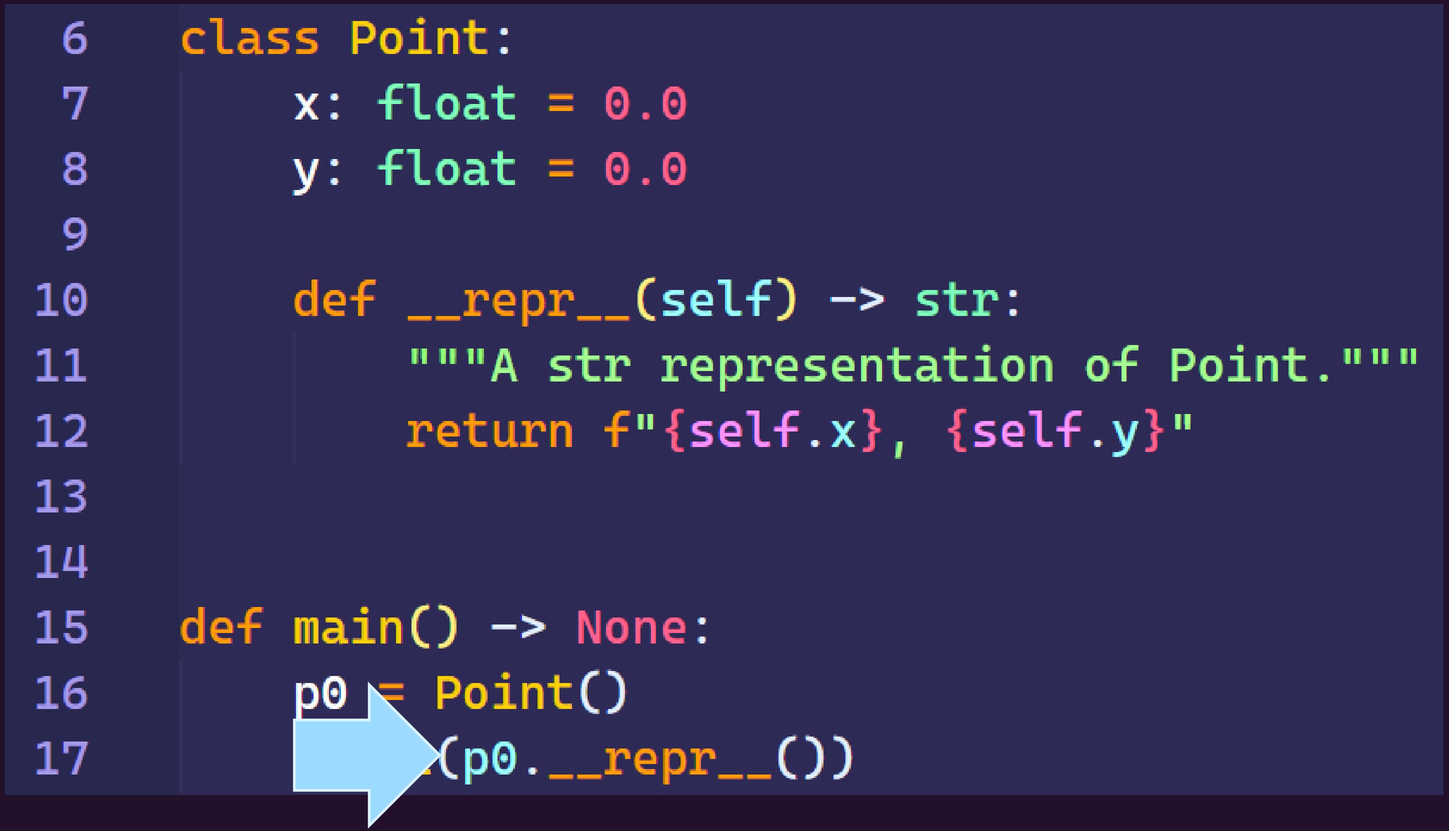
p0

Point

x 0.0

y 0.0

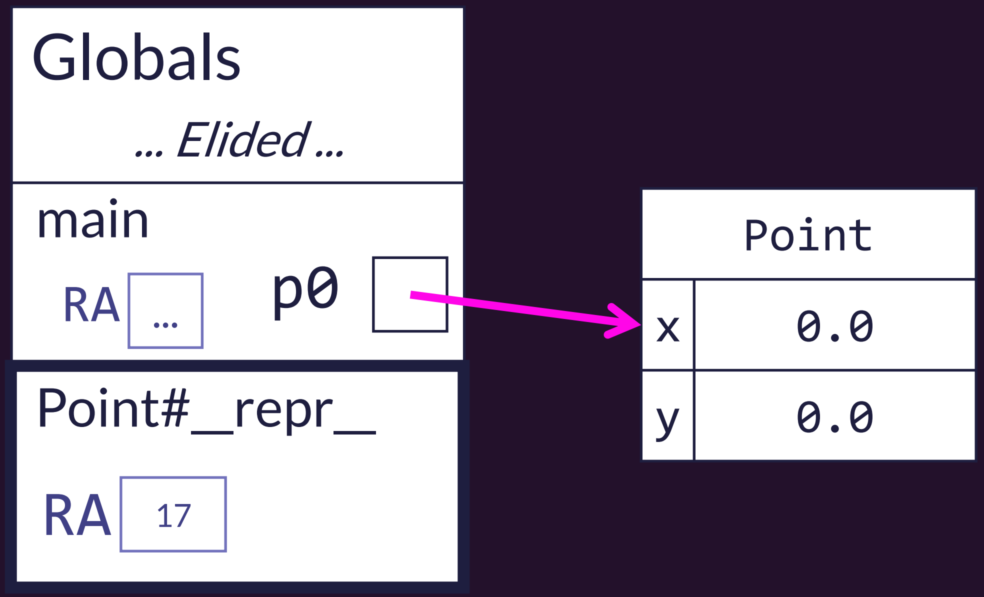
How is this *method call* processed? First, a frame is added...



```python
 6  class Point:
 7      x: float = 0.0
 8      y: float = 0.0
 9
10      def __repr__(self) -> str:
11          """A str representation of Point."""
12          return f"{self.x}, {self.y}"
13
14
15  def main() -> None:
16      p0 = Point()
17      (p0.__repr__())
```

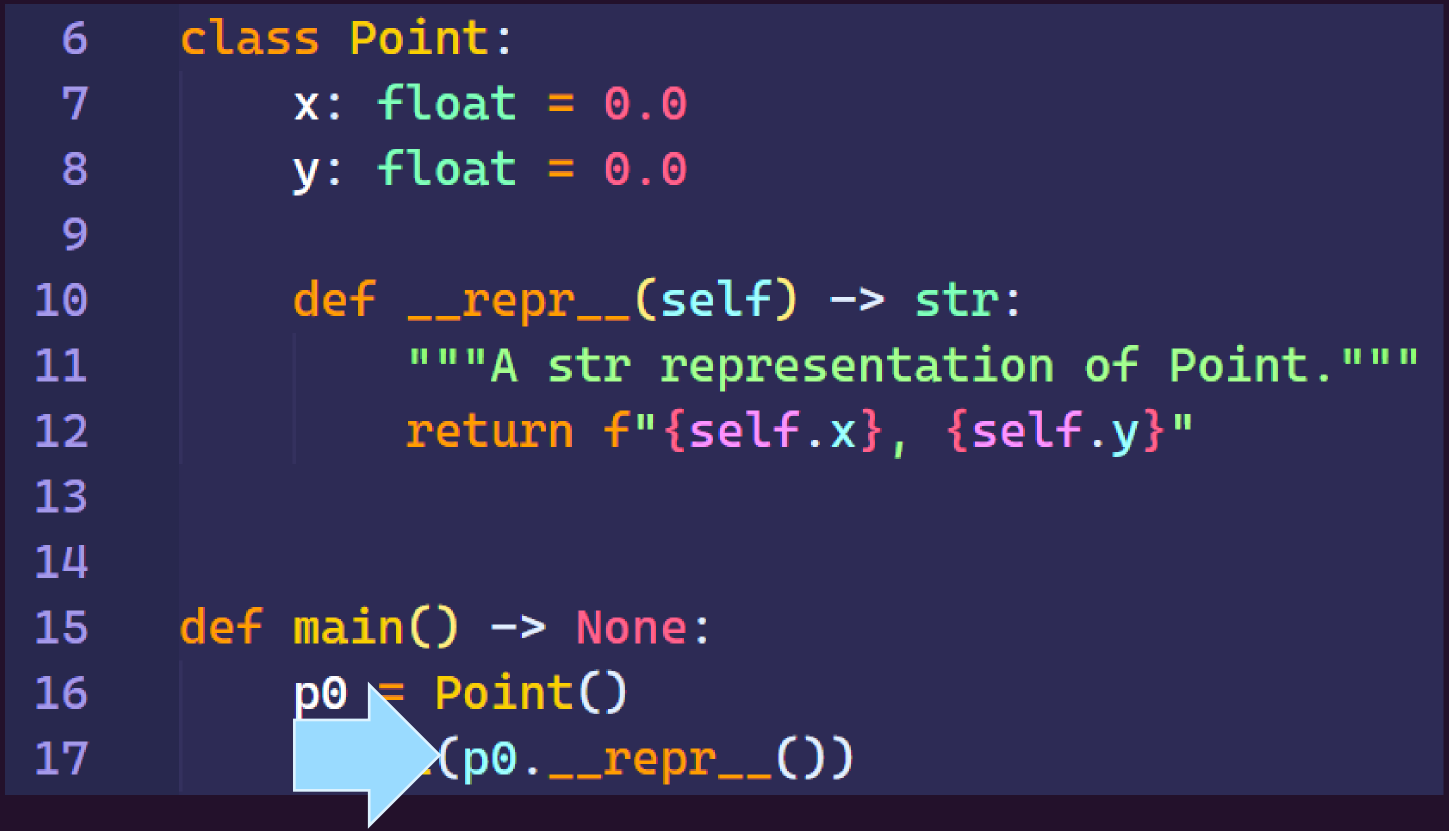
The Stack

The Heap

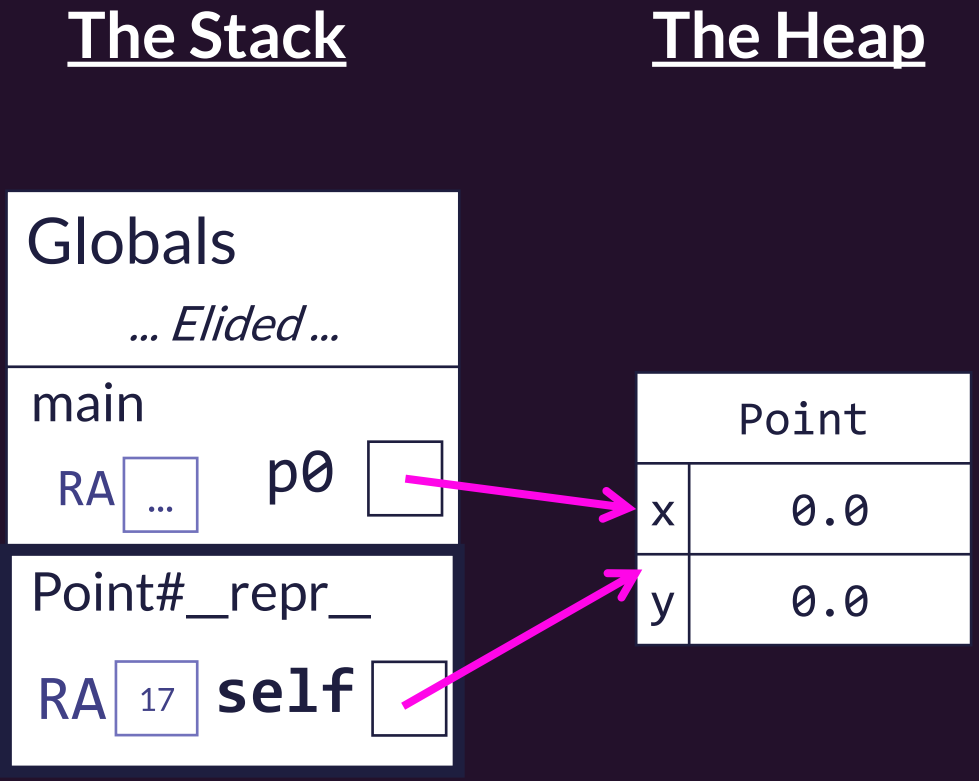


What's up with this pound sign? It's conventional across many programming languages to identify a method by **`ClassName#method`**.

THEN, a reference named **this** is established TO the object the method was called on.... and *this* is *all the magic* of a **method call**.

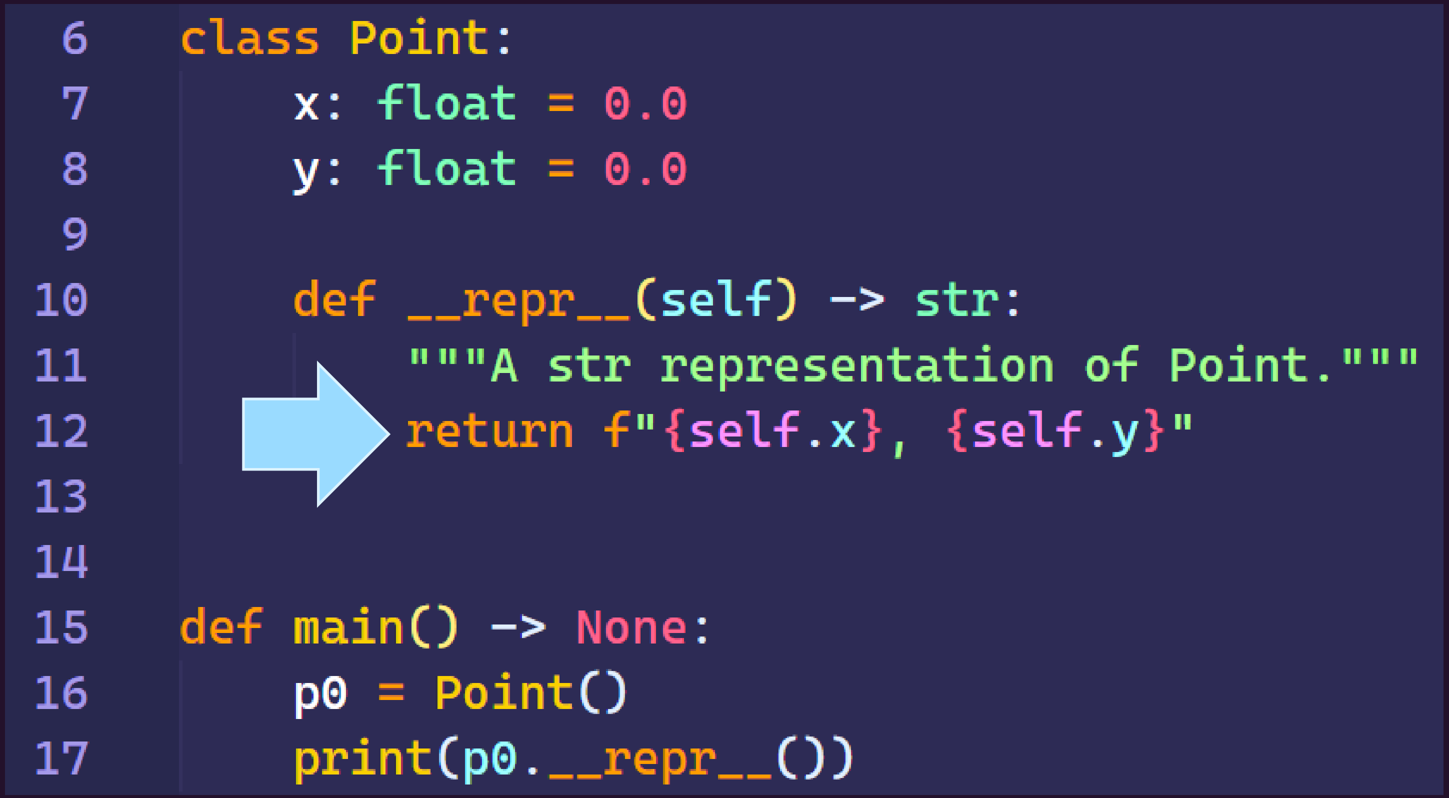


```
class Point:
    x: float = 0.0
    y: float = 0.0

    def __repr__(self) -> str:
        """A str representation of Point."""
        return f"{self.x}, {self.y}"


def main() -> None:
    p0 = Point()
        (p0.__repr__())
```

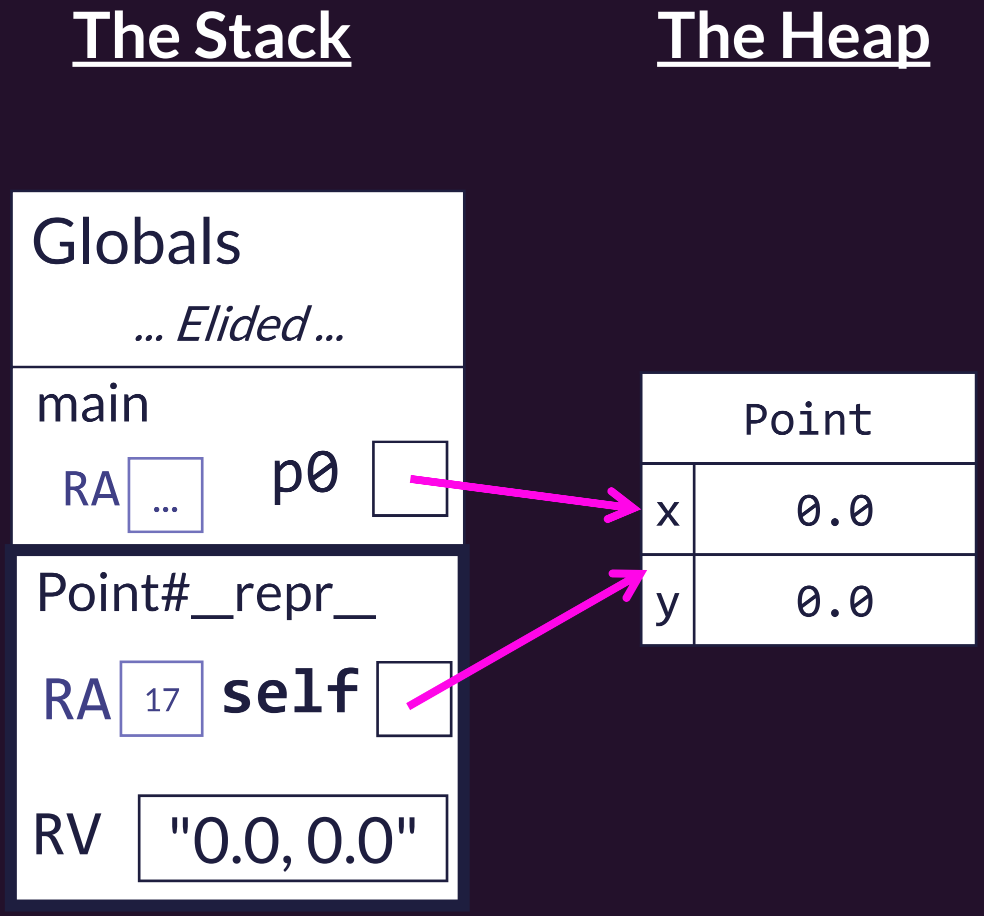


What's up with this pound sign? It's conventional across many programming languages to identify a method by **ClassName#method**.

In the method call evaluation, notice *self* refers to the same object the method was called on.



```
 6  class Point:
 7      x: float = 0.0
 8      y: float = 0.0
 9
10      def __repr__(self) -> str:
11          """A str representation of Point."""
12          return f"{self.x}, {self.y}"
13
14
15  def main() -> None:
16      p0 = Point()
17      print(p0.__repr__())
```



The Stack

The Heap

Globals

... Elided ...

main

RA ...

p0

Point#__repr__

RA 17

self

RV "0.0, 0.0"

Point	
x	0.0
y	0.0

Method Call Tracing Steps

When a method call is encountered on an object,

1. The processor will determine the class of the object and then confirm it:
    1. Has the method being called defined in it.
    2. The method call's arguments agree with the method's parameters.
2. Next it will initialize the RA, parameters, *and* the **`self`** parameter
    - The *first parameter* is assigned a reference to the object the method is called on
    - The *first parameter* of a method is idiomatically named **`self`** in Python
3. Finally, when the method completes, processor returns to the RA.

Why have both functions and methods?

- Methods allow objects to have "built-in" functionality
  - You don't need to import extra functions to work with an object, they are bundled.
  - As programs grow in size, methods and OOP have some additional features to help teams of programmers avoid accidental errors.

- Different schools of thought in *functional programming-style (FP)* versus *object-oriented programming-style (OOP).*
  - *Both are equally* ***capable****, but some problems are better suited for one style vs. other.*

- FP tends to shine with *data processing* problems
  - Data analysis programs like processing *stats* and are natural fits

- OOP is great for stateful systems like *user interfaces, simulations, graphics*