Constructors & Methods

Object-oriented Programming

- So far, you've used objects as compound data types
  - i.e. to model the *attributes* of a Pizza

- You've written functions, *separate from classes*, that take in objects

- **Object-oriented Programming** concepts build on the concept of *classes*

  1. **Methods** allow you to give all objects special *capabilities*

  2. **Constructors** allow you to fully initialize objects *before their use*

Review of Classes and Objects

- A class defines a new **Data Type**
    - The class definition specifies properties

- *Instances* of a class are called **objects**
    - To create an object you must call its constructor: `ClassName()`

- *Every object of a class* has the **same attributes,** but with **its own values**

- Objects are **reference-types**
    - Variables do not hold objects, but rather ***references to objects***

Follow-along: Simple Method App

- Let's implement and call the say_hello method

```python
"""An example of methods."""


class Person:

    ...  # attributes elided

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


def main() -> None:
    """Entrypoint of program."""
    a_person: Person = Person()
    a_person.say_hello()


if __name__ == "__main__":
    main()
```

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.**

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

2. Once defined, we can then call it.

```python
say_hello()
```

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

```python
class Person:

    ...   # attributes elided

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

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

```python
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, initializing its name attribute, and also calling its say_hello method.

5. Check-in on PollEverywhere

```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()`

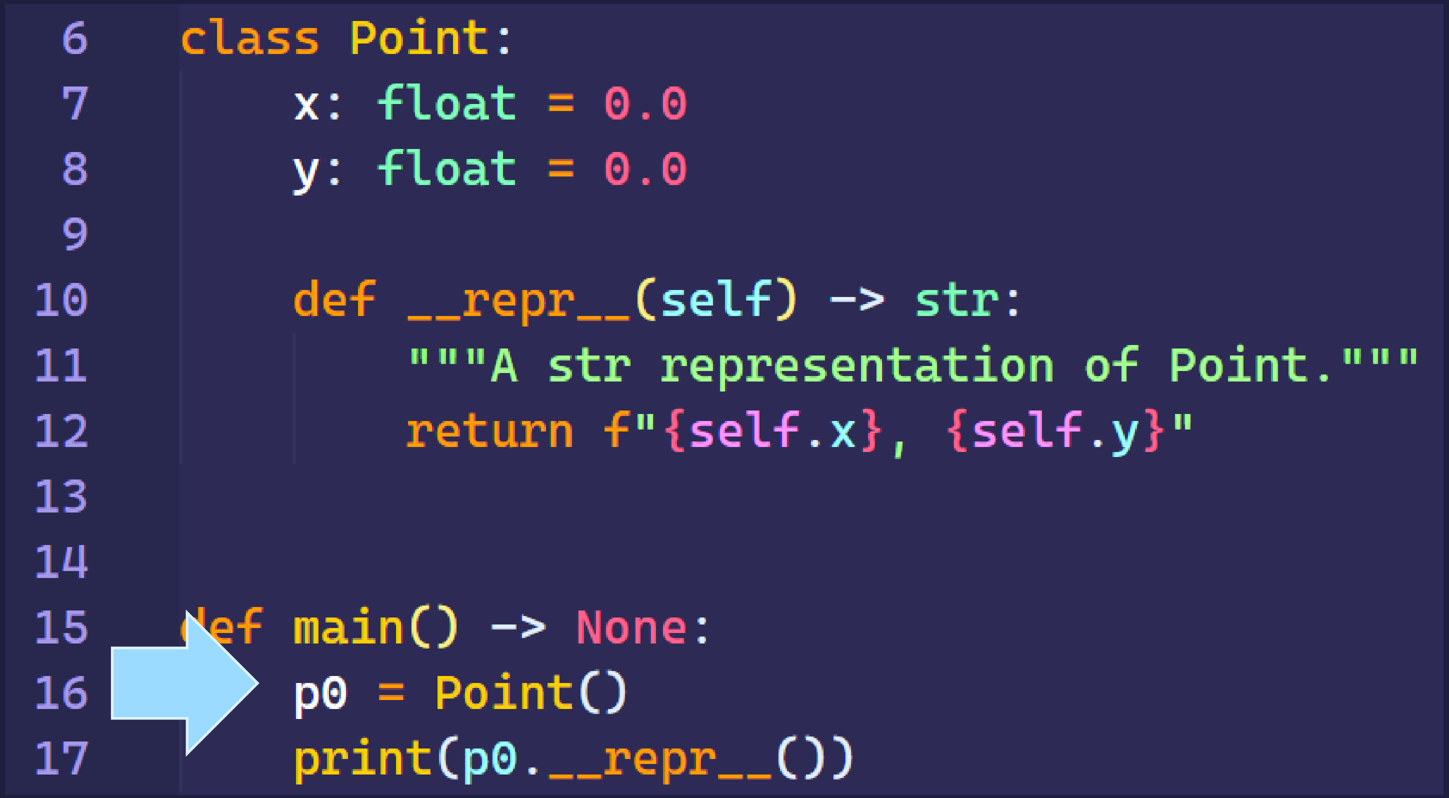
- The say_hello method's definition is:

```
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.

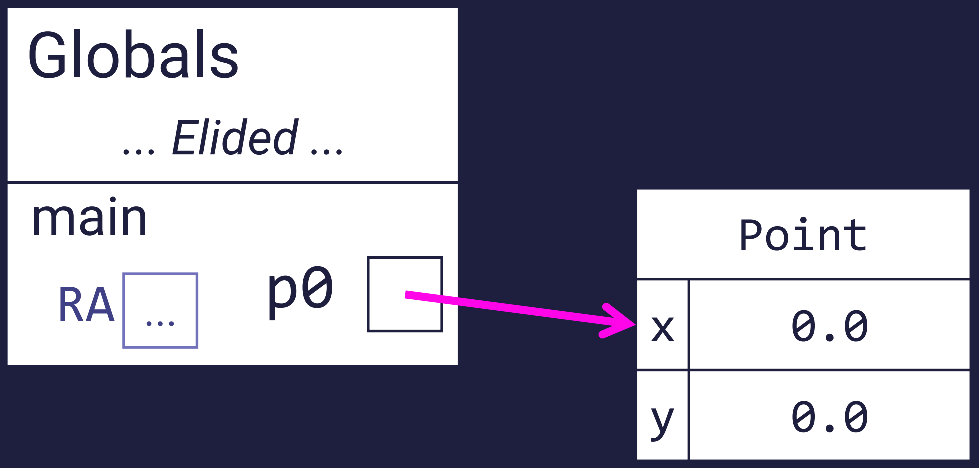
Suppose the interpretter *just* completed this line...



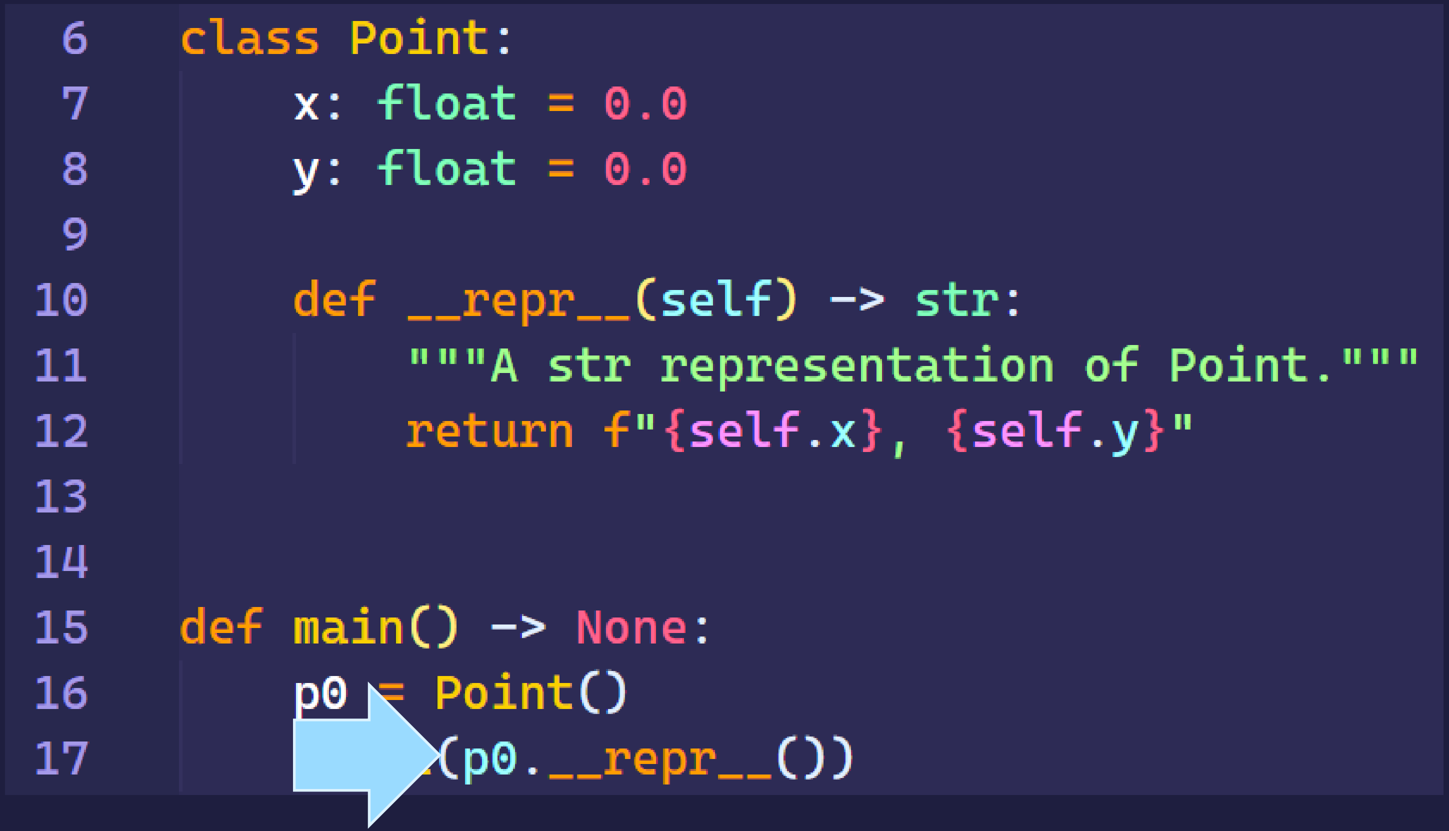
```python
class Point:
    x: float = 0.0
    y: float = 0.0

    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

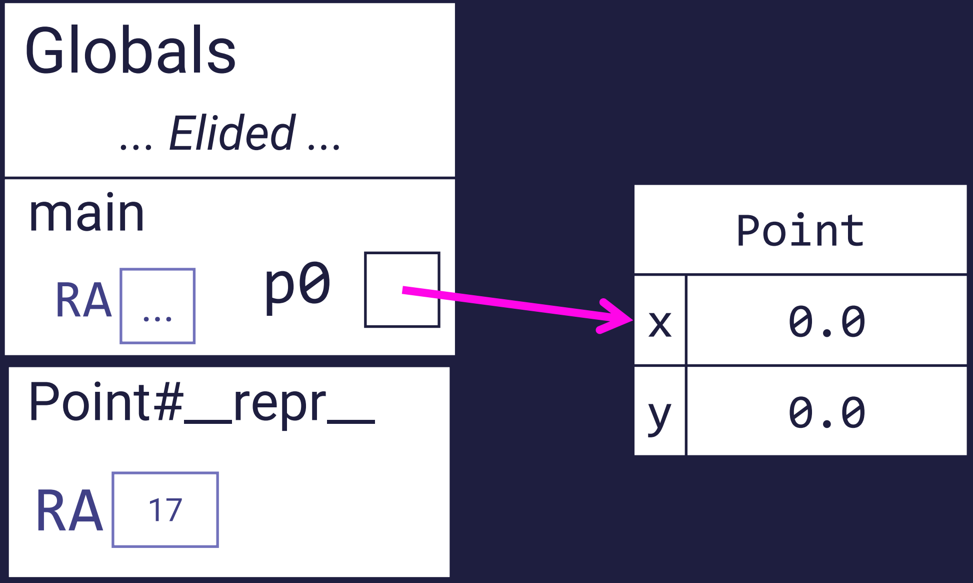


How is this *method call* processed? First, a frame is added...



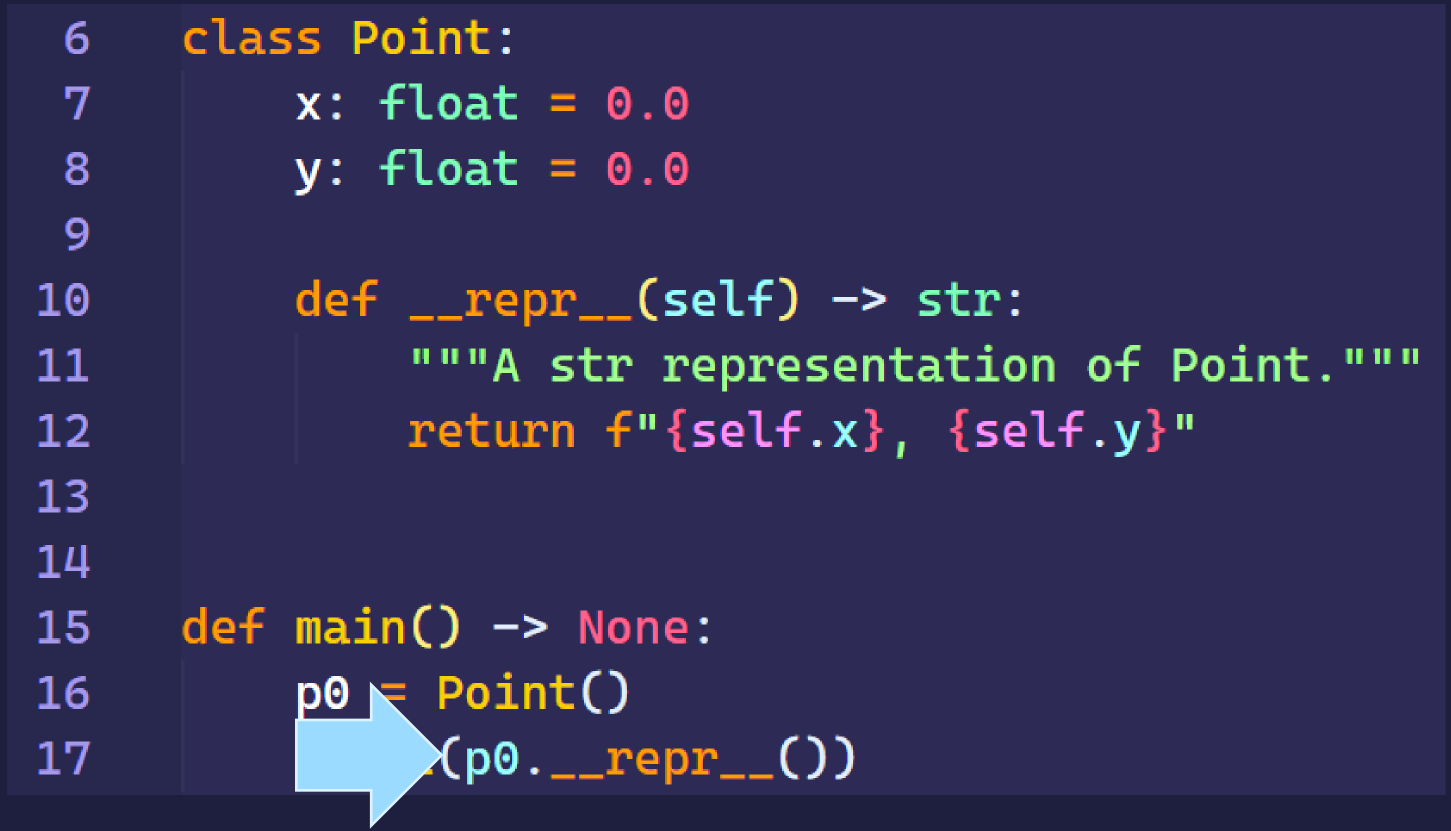
```
class Point:
    x: float = 0.0
    y: float = 0.0

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


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

The Stack **The Heap**

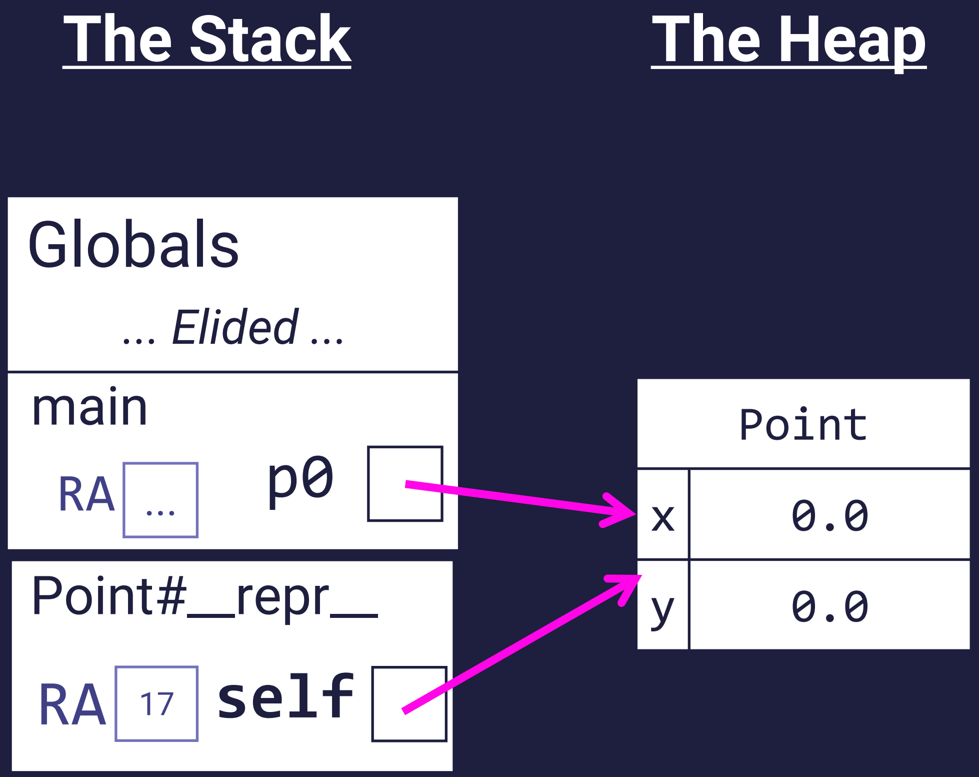


What's up with this pound sign? It's conventional across many programming languages to identify a method by `ClassName#method`.

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__())
```



The Stack

The Heap

Globals

... Elided ...

main

RA ...

p0

Point#__repr__

RA 17

self

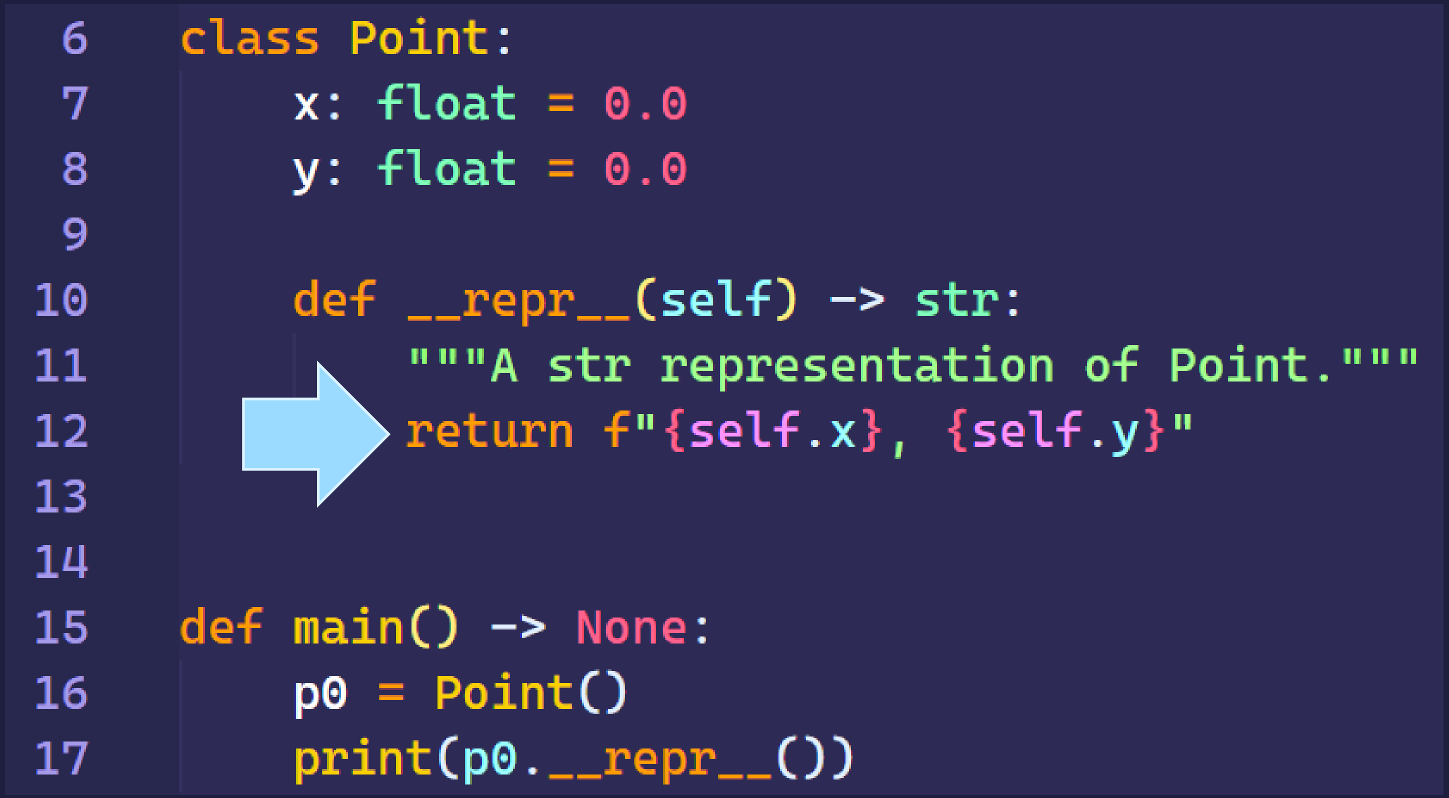
Point

x 0.0

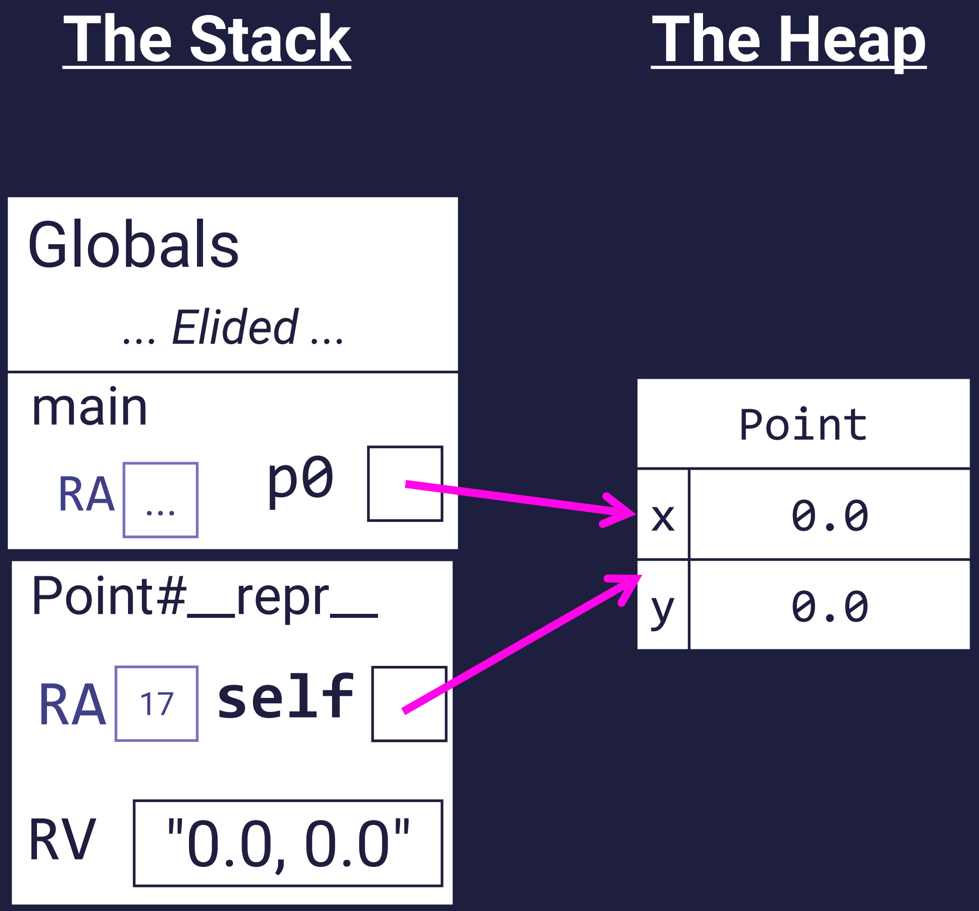
y 0.0

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.



```
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__())
```



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.

Hands-on: Practice with self

- In ls35_constructor.py, add the code right
- let's make it easy to move a Point relative to its current position.

1. Declare a method of Point named translate.
   - two parameters: dx and dy
   - returns None
   - method body should increase the point object's x and y attributes by dx and dy, respectively
2. Call **`translate`** on **`Point p0`** in the `main` function using any values you'd like, before printing
3. Once you've tried that it works, check-in on PollEv.com/compunc

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

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


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


if __name__ == "__main__":
    main()
```

Why have both functions and methods?

- 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*

- 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.

Constructors

- An object's attributes must be initialized before the object is usable

- A constructor allows you to
  1. Specify initial values of attributes upon creation of an object
  2. Require certain attributes be decided by the caller of the constructor

- A constructor is just a *magic* method
  - Dunder-name is **`__init__`**
  - Also has a first parameter named **`self`**
  - Return type is omitted

- A class' constructor is *automagically* called each time the **`Classname()`** call expression is evaluated.
  - "Magic" method because you do not call it directly. Notice you never call __init__() anywhere. The language calls it in its evaluation of construction.

Before

```
a = Point()
a.x = 10;
a.y = 0;
```

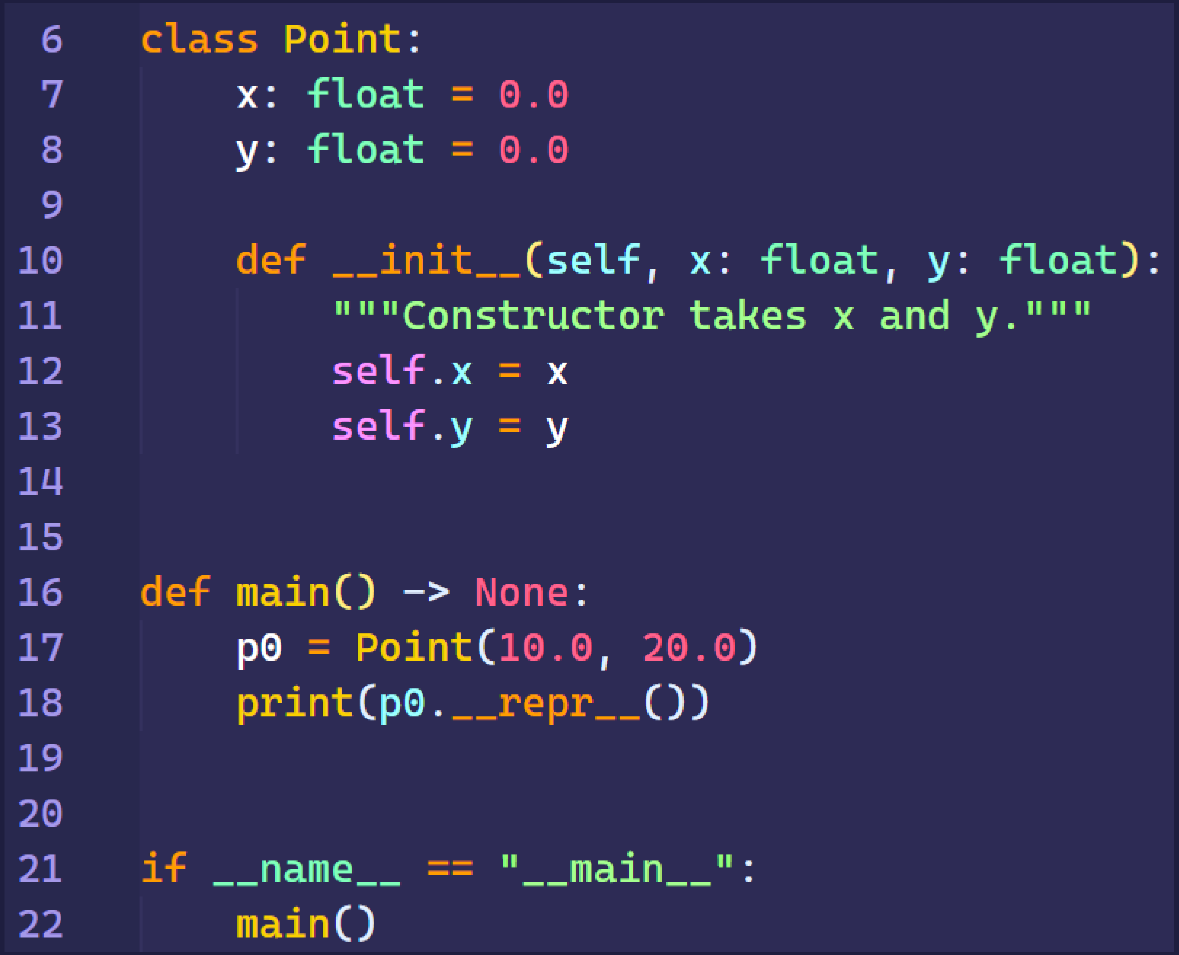
Defining a constructor

```
class Point:

    x: float
    y: float

    def __init__(self, x: float, y: float):
        self.x = x
        self.y = y
```

After

```
a = Point(10, 0)
```

Diagram Example



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

    def __init__(self, x: float, y: float):
        """Constructor takes x and y."""
        self.x = x
        self.y = y


def main() -> None:
    p0 = Point(10.0, 20.0)
    print(p0.__repr__())


if __name__ == "__main__":
    main()
```