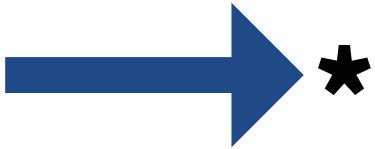
POINTERS IN PYTHON

TABLE OF CONTENTS

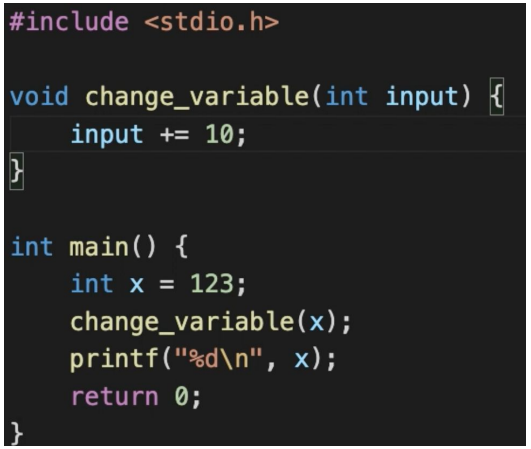
Pointers: What's the Point?

- Pointers are essential to writing efficient code in many languages.
- C and C++ rely heavily on pointers. Many libraries and language features require them!
- In the next two videos, you'll learn everything you need to know about pointers to follow along with this course.



Pointers: What's the Point?

- As you follow along, you'll see some C code that looks something like this:



```c
#include <stdio.h>

void change_variable(int input) {
    input += 10;
}

int main() {
    int x = 123;
    change_variable(x);
    printf("%d\n", x);
    return 0;
}
```

- I'll explain what code like this does, and how it relates to pointers.

Looking Ahead

1. You'll learn why Python doesn't support the idea of pointers.
   - Look under the hood at variables and how Python manages memory.
2. You'll learn how to simulate pointer behavior in Python.
3. You'll learn how to create C-style pointers with the `ctypes` module.
   - This will allow you to work with C libraries from within Python.

- Even though Python doesn't explicitly use pointers, understanding how they work will make you a better programmer.

Pointers: A Real-World Example

- You just had built a new office space, but the sign with the logo is wrong!

Pointers: A Real-World Example

- You contact a company who specializes in this. They'll fix the sign, but they need an office building to fix first! They give you their address.

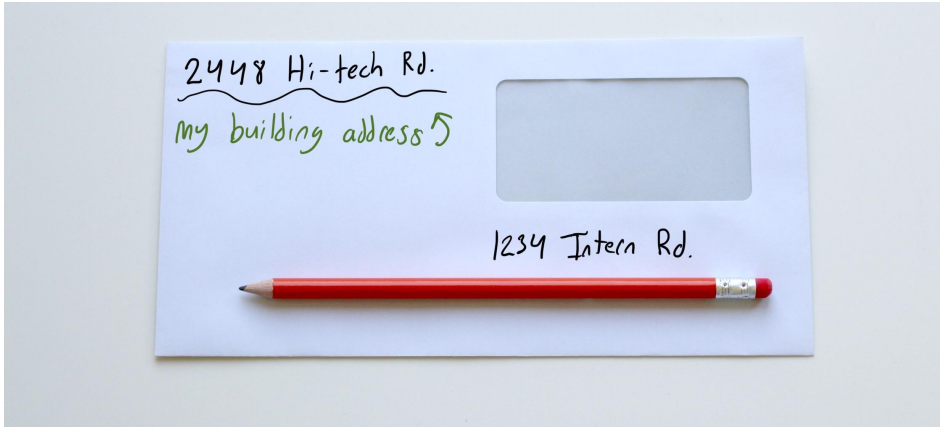
Pointers: A Real-World Example

- Your first thought is to simply send them a copy of the ACTUAL BUILDING to fix. But that's not practical for many reasons.

- In programming terms, this is like copying a variable that occupies lots of space in memory. It's slow and requires lots of memory!

Pointers: A Real-World Example

- Instead, you ask the company if you can just send them your office address. They agree, so you send a letter in the mail with the address on it.



- This is the equivalent of passing a large variable by reference. You pass a copy of the memory address, not the data that lives at that address.

Pointers: A Real-World Example

- You took a copy of your office address and stored it in an envelope. This is a pointer.
- You send this envelope off to the company, effectively passing your office by reference.
  - Passing by value would mean you send a copy of the actual office. Bad!
- They follow the address you sent and fix your office sign directly! No need to return any office building to you.

Pointers: A Real-World Example

- That’s how pointers are used for efficiency.
- If used right, they can reduce the overall memory usage of your program.
  - This is useful for devices with very little memory, such as microcontrollers.

- But if pointers are so great, then why doesn’t Python use them?

Why Doesn’t Python Have Pointers?

- Out of the box, Python does not support creating and using pointers to reference data in memory.
- Pointers go against the *Zen of Python*.
  - They are often messy to work with.
  - They encourage implicit changes instead of explicit ones.
  - Code becomes harder to read with * and & everywhere.
    - Does this variable contain data I want, or is it a pointer to it?
  - They beg for you to do something dangerous, like read from a section of memory you aren’t supposed to.

Why Doesn’t Python Have Pointers?

- Python tends to abstract away from low-level details like memory management.
- You don’t have to worry about memory management when passing variables into functions.
- Automatic processes like garbage collection ensure any variable that’s no longer accessible is freed from memory.

Why Doesn’t Python Have Pointers?

Python	C
• Faster and easier to write. You only have to worry about the scope of variables. • Slower to run. Processes like garbage collection result in slower execution time and increased memory usage. • Good for high-level programs like web apps, utilities, high-level game logic, graphing, etc.	• Slower to write. You must know how to properly pass variables using pointers (when required). • Faster to run. Since you manage memory manually, no garbage collection process is needed. Variables are never in memory longer than they need to be. • Good for low-level programs like drivers, game engines, operating systems, etc.

Immutable vs. Mutable Objects

- A **mutable** object can have its value changed after it has been created.
- For example, the list:

```
values = [1,2,3]
values += [4]
```

- The list is a mutable type. We can add values to it, and under-the-hood Python can add to the data in the object without re-creating the object from scratch.

Immutable vs. Mutable Objects

- An **immutable** object must be re-created when modified.
- This results in a new copy of the object which replaces the old one.
- For example, the string:

```
name = 'Python'
name = 'Real ' + name
```

- `str` is an immutable type. When we modify it, we must store the new, modified copy in a variable. This is why any string methods that modify it return a copy of the modified string.

Immutable vs. Mutable Types

Type	Immutable?
int	Yes
float	Yes
bool	Yes
complex	Yes
tuple	Yes
frozenset	Yes
str	Yes
list	No
set	No
dict	No

Variables in C

- Variables in C can be declared and initialized like this:

```c
int x = 2337;
```

- Three things happen under-the-hood:
  1. Memory is allocated for an integer
  2. 2337 is stored in that memory space
  3. **x** is mapped to that memory

X	
Location	0x7f1
Value	2337

Variables in C

- We can change the value of **x** like this:

```
x = 2338;
```

- Integers in C are mutable types. We can change the value of **x**, and it will override the previous value without having to use any additional memory (no duplication!)

X	
Location	0x7f1
Value	2338

Variables in C

- We can copy the value of **x** into a new variable **y**:

```
int y = x;
```

X			Y	
Location	0x7f1		**Location**	0x7f5
Value	2338		**Value**	2338

- Because the **value** of **x** is copied into **y**, the values can be modified independently of one another. A change in one does not reflect in the other.

Variables in Python

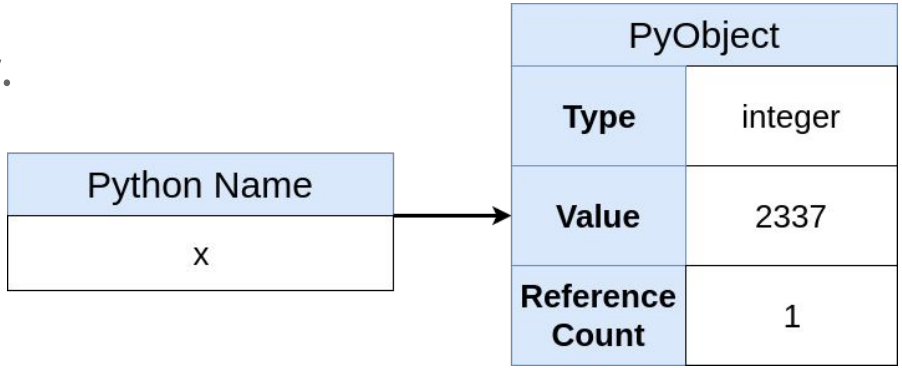
- Technically, Python does not have variables. It has **names**.
- This is a pedantic point, but it matters when dealing with memory.

Variables in Python

- This Python code assigns the name x to the value 2337:

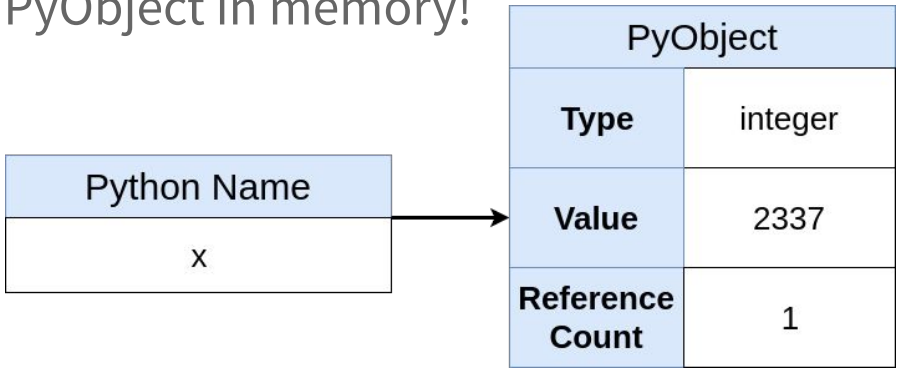
```
x = 2337
```

- Under-the-hood:
  1. A PyObject is created in memory.
  2. PyObject type is set to *integer*.
  3. PyObject value is set to 2337.
  4. A name **x** is created.
  5. **x** is set to point to the PyObject.
  6. PyObject reference count is set to 1.



Variables in Python

- A PyObject is different than the Python `object` type.
- PyObject is the CPython implementation of the Python `object`, implemented using a C struct (a data structure that groups related variables together).
- Since every Python type inherits from `object`, every Python object is represented as a PyObject in memory!

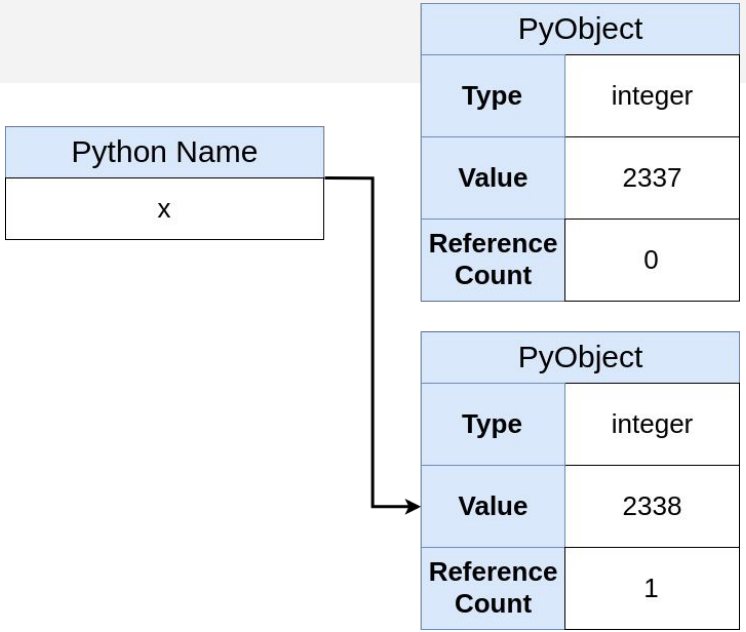


Variables in Python

- Changing what ‘x’ points to is a little more complex in Python

```
x = 2338
```

- Under-the-hood:
  1. A new PyObject has to be created because integers are immutable.
  2. The **x** name points to the new PyObject.
  3. Reference counts for both PyObjects are updated.

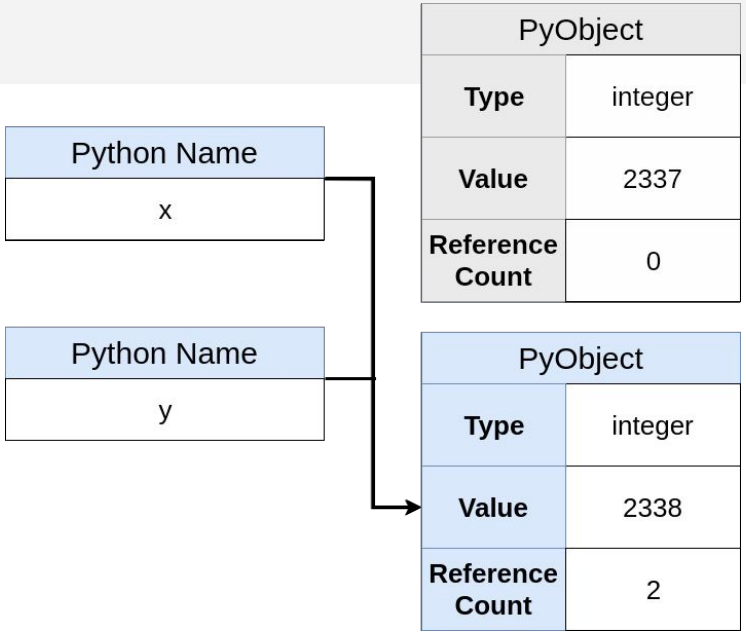


Variables in Python

- What happens if we try to set a new name **y** to the value of **x**?

```
y = x
```

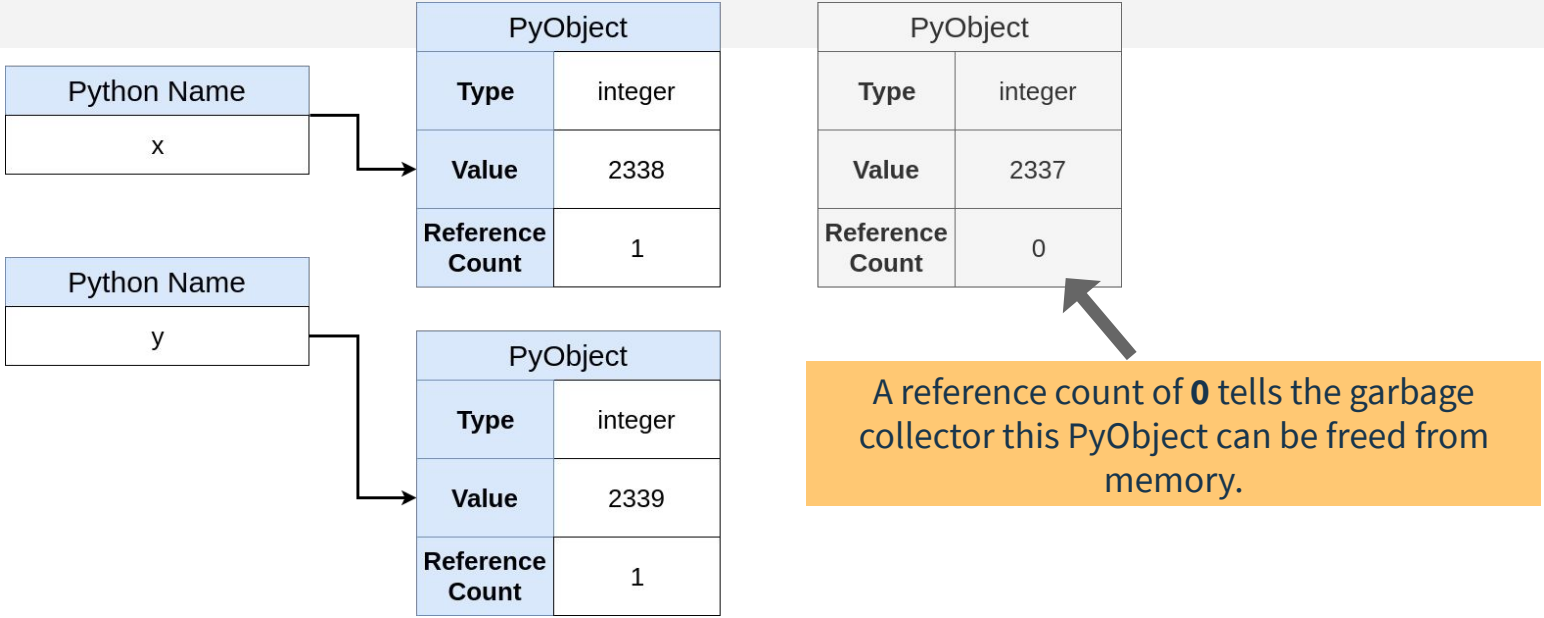
- Under-the-hood:
  1. A new name **y** is created and pointed to the same PyObject.
  2. The original PyObject has a reference count of 0, telling the garbage collector that it can be freed from memory.



Variables in Python

- Integers are still immutable, so incrementing y still requires duplication.

```
y += 1
```



Variables in Python

- In Python, we assign names to spaces in memory called PyObjects.
- Modifying an immutable type requires duplicating the PyObject and changing the name to point to this new PyObject.
- Modifying a mutable type does not require duplication because the value of the PyObject can be modified directly.

Intern Objects

- If you were thinking all this duplication is unnecessarily inefficient, you're right!
- The core Python developers knew this, so they created what are called **intern objects.**
- These are pre-created PyObjects in memory that can be accessed from anywhere. They never need to be freed.
- Before creating a new PyObject, Python checks to see if a suitable intern object already exist.
- Think of these objects like a cache for the most commonly used PyObjects.

Intern Objects

- You can manually intern objects, but Python interns some by default.
- What gets interned is implementation-specific, but CPython 3.7 interns:
  1. Integers between -5 and 256
  2. Strings that are less than 20 characters long and contain only ASCII letters, digits, or underscores.
- These sets were chosen because they are the most commonly used in Python programs.

Interned Integers

```
>>> x = 10
>>> y = 10
>>> x is y
True
>>> id(x)
1557554416
>>> id(y)
1557554416
```

Interned

```
>>> x = 1500
>>> y = 1500
>>> x is y
False
>>> id(x)
30530816
>>> id(y)
30530800
```

Not Interned

Interned Strings

```
>>> s1 = 'realpython'
>>> s2 = 'realpython'
>>> s1 is s2
True
>>> id(s1)
31272336
>>> id(s2)
31272336
```

Interned

```
>>> s1 = 'realpython!'
>>> s2 = 'realpython!'
>>> s1 is s2
False
>>> id(s1)
31272536
>>> id(s2)
31272576
```

Not Interned

Intern Objects

- What gets interned is implementation specific.
- Interning behavior may differ between Python scripts and the Python interactive shell.
    - Compilers used to compile scripts are often smart enough to intern different names if they detect they are pointing to the same values, even if they fall outside the range for pre-interned objects.

Pass-by-Value vs. Pass-by-Reference

- **Pass-by-Value:**
  - Pass a copy of the value to the receiving function.
  - This requires the object be duplicated in memory, and the receiving function will only be able to access this new memory space.
- **Pass-by-Reference:**
  - Pass a copy of the memory address of the object (a pointer.)
  - This is often much smaller than the actual object.
  - The receiving function can dereference (follow) that memory address to access or modify the value there directly.

Pass-by-Value vs. Pass-by-Reference

- So which one does Python use when passing names to functions?
  - Neither!
- Python uses what it calls **pass-by-assignment**, meaning the parameters in the function being called point to the objects in memory that were passed into them as arguments.
  - This looks like pass-by-reference, but we're not passing in a memory address to a value. We're passing the address of a PyObject, which itself contains the desired value.
  - Instead of passing an address to the data, we pass an address to data which *contains* the data.

Pass-by-Value vs. Pass-by-Reference

- We're passing in a reference to a PyObject in memory, which as you learned behaves differently depending on if the object is mutable or immutable.
- Passing an **immutable** type acts like **pass-by-value** because you cannot directly modify the PyObject being passed into the function. It must be duplicated, then the modified PyObject can be returned from the function.
- Passing a **mutable** type acts like **pass-by-reference** because you can directly modify the PyObject being passed into the function without duplication.
    - However, if any new PyObjects are created in the function, they are still limited to the scope of the function. They'll need to be returned from the function to be used outside that scope.

The `ctypes` Module

- You may need to utilize a C library in your Python code.
- Or maybe you want to write a performance-critical portion in C, then interface with it from Python.
- The built-in `ctypes` module allows for this!
  - Provides all the necessary Python required to interface with C libraries.
  - This is probably the closest thing you'll get to *real* C-style pointers in Python.

Conclusion

- In this course you learned:
  1. How pointers work in C
  2. What mutable and immutable objects are in Python
  3. How to utilize mutable objects and classes to mimic pointer behavior
  4. How Python code can interface with C code using `ctypes`
- Using what you've learned, you can write Python libraries that are easier to work with. Or, you can break free from the performance limits of Python by writing performance-critical sections of Python code in C.