Overview

The Python object model

Named references to objects

Value vs. identity equality

Passing arguments and returning values

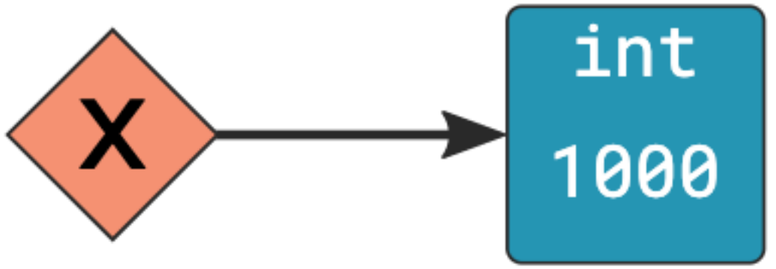
Python's type system

Scopes to limit name access

"Everything is an object"

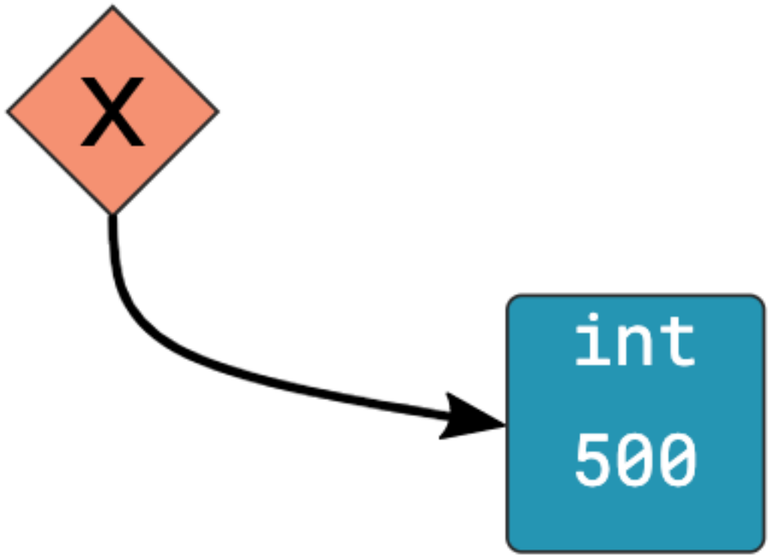
Assigning to a Variable

```
x = 1000
```



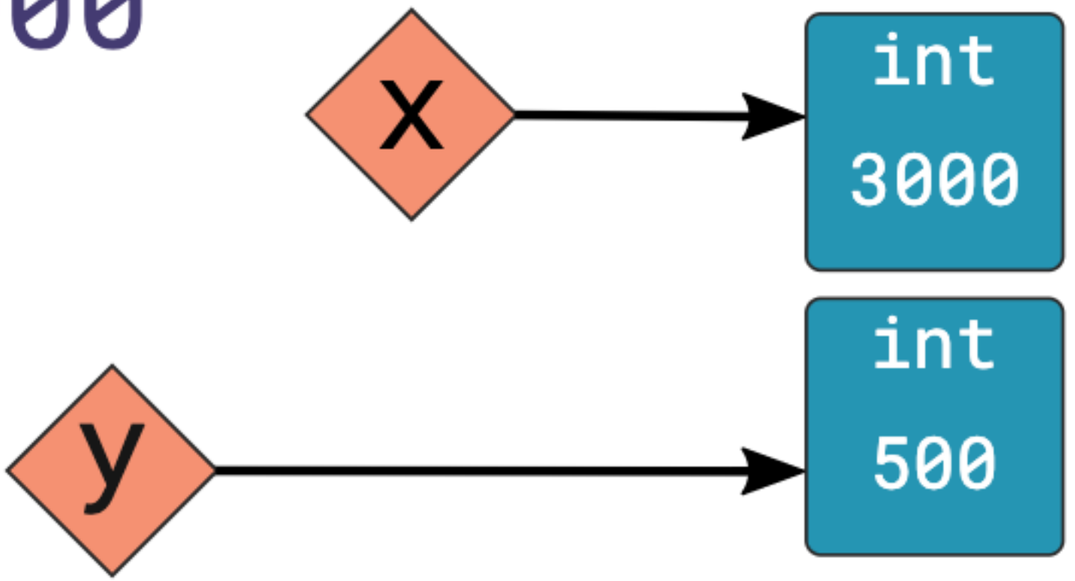
Reassigning a Variable

```
x = 500
```



Multiple Assignment

```
y = x
x = 3000
```



`id()`

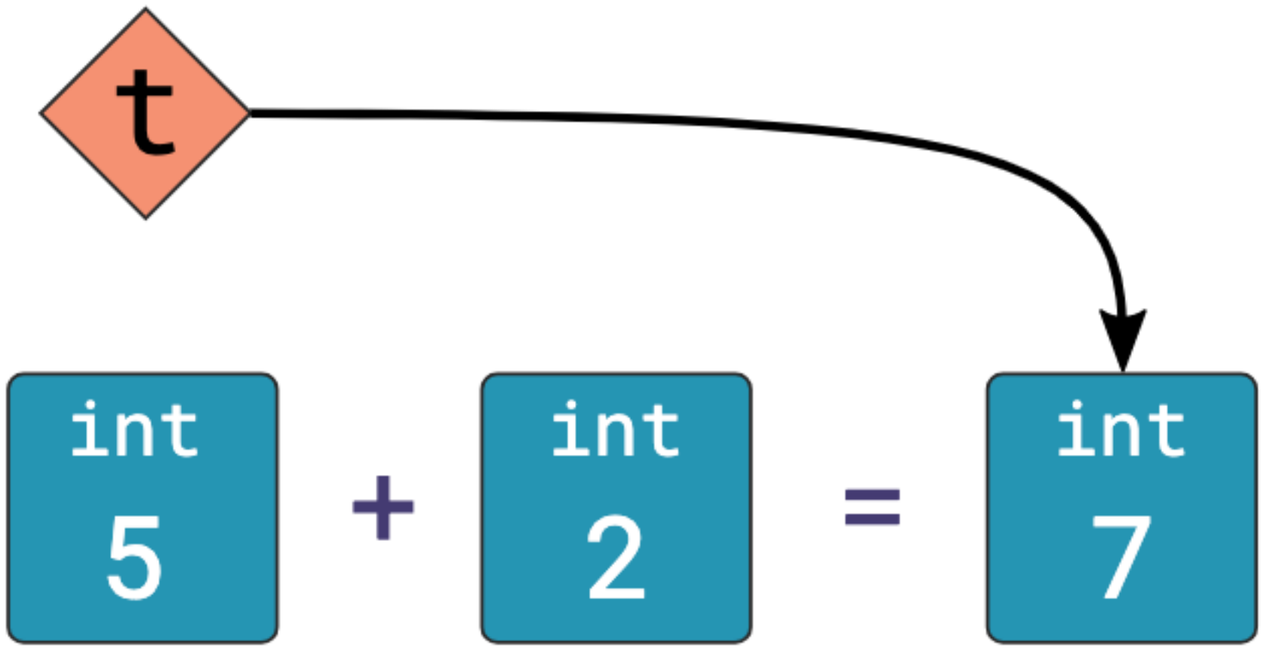
Returns a unique integer identifier for an object that is constant for the life of the object.

id()

```
>>> a = 496
>>> id(a)
4301474256
>>> b = 1729
>>> id(b)
4301475248
>>> b = a
>>> id(b)
4301474256
>>> id(a) == id(b)
True
>>> a is b
True
>>> a is None
False
>>> t = 5
>>> id(t)
4298278608
>>> t += 2
>>> id(t)
4298278672
>>>
```

Augmented Assignment

```
t = 5
t += 2
```

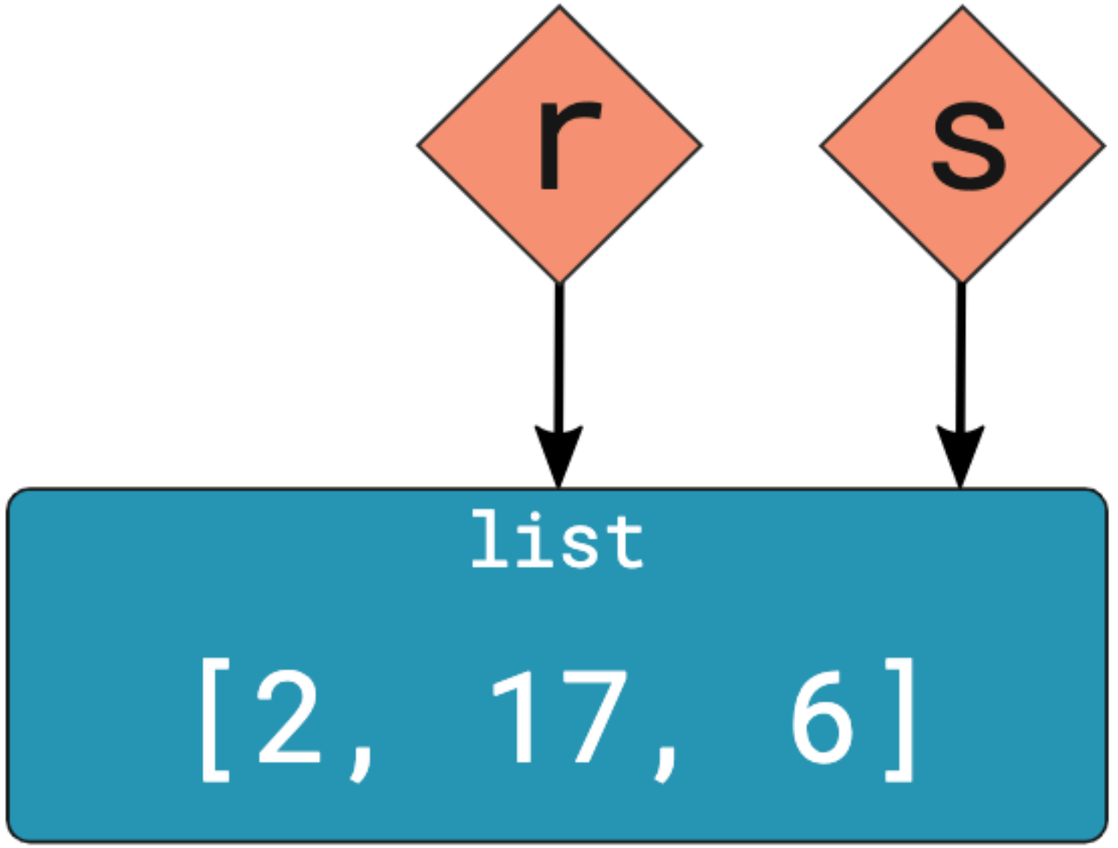


Mutable Objects

```
>>> r = [2, 4, 6]
>>> r
[2, 4, 6]
>>> s = r
>>> s
[2, 4, 6]
>>> s[1] = 17
>>> s
[2, 17, 6]
>>> r
[2, 17, 6]
>>> s is r
True
>>>
```

Mutable Objects

```
r = [2, 4 ,6]
s = r
s[1] = 17
s is r
```



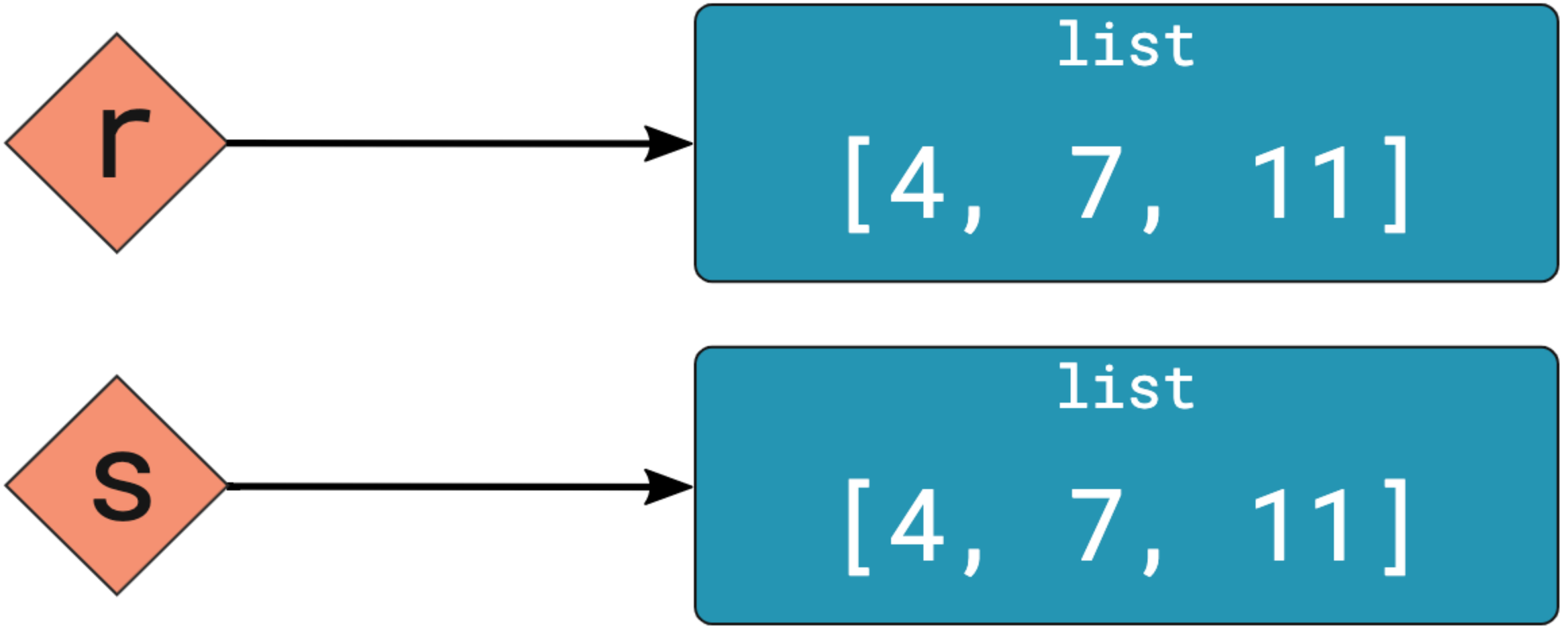
Python doesn't have variables in the sense of boxes holding a value.

Python has named references to objects.

Value vs. Identity Equality

```
>>> p = [4, 7, 11]
>>> q = [4, 7, 11]
>>> p == q
True
>>> p is q
False
>>> p == p
True
>>>
```

Value Equality



Value vs. Identity Equality

Value-equality and identity equality are fundamentally different concepts.

Comparison by value can be controlled programatically.

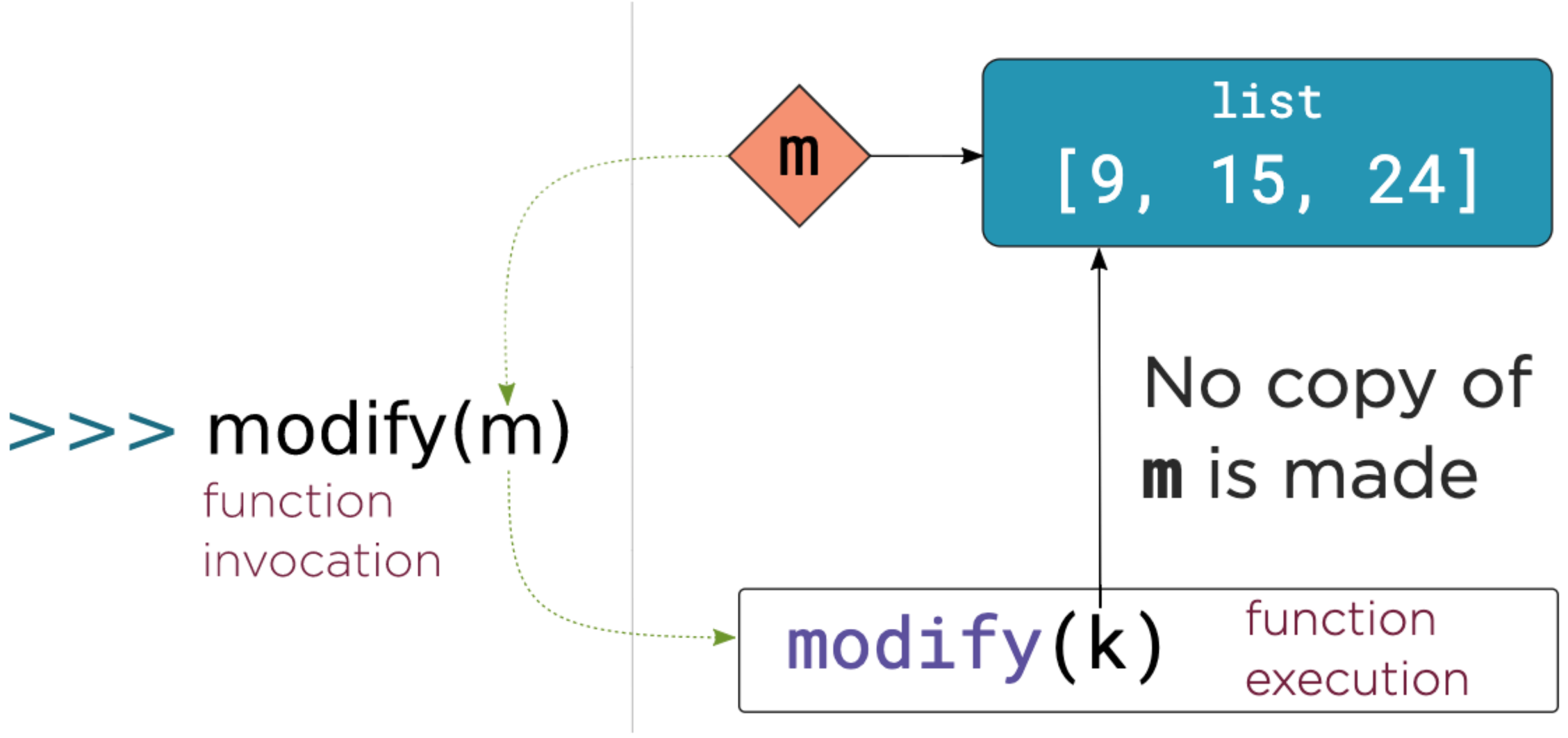
Identify comparison is unalterably defined by the language.

Passing Arguments and Returning Values

Argument Passing

```
>>> m = [9, 15, 24]
>>> def modify(k):
...         k.append(39)
...         print("k =", k)
...
>>> modify(m)
k = [9, 15, 24, 39]
>>> m
[9, 15, 24, 39]
>>>
```

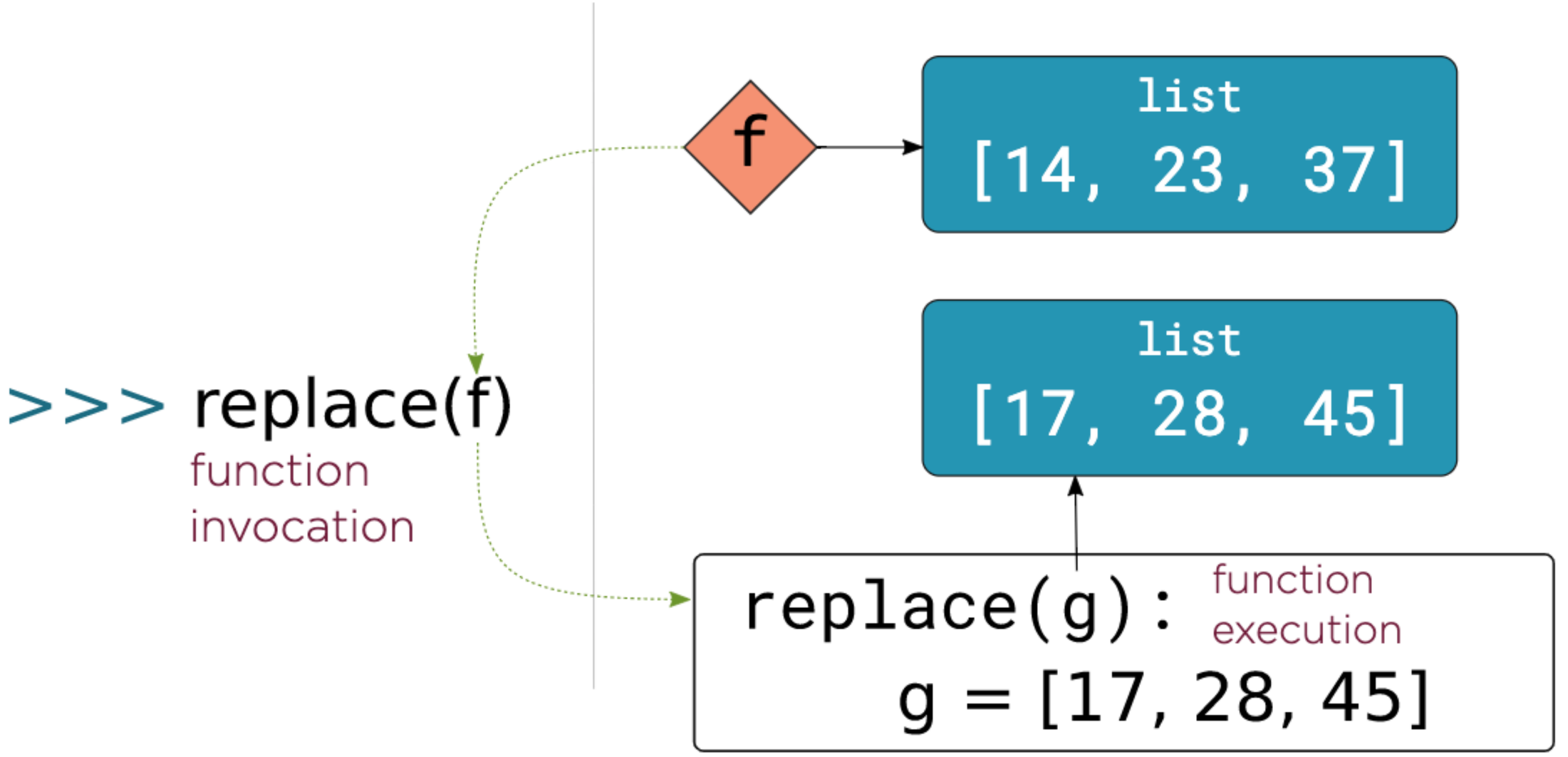
Argument Passing Semantics



Replacing Argument Value

```
>>> f = [14, 23, 37]
>>> def replace(g):
...         g = [17, 28, 45]
...         print("g =", g)
...
>>> replace(f)
g = [17, 28, 45]
>>> f
[14, 23, 37]
>>>
```

Replacing Argument Value



Mutable Arguments

```
>>> def replace_contents(g):
...     g[0] = 17
...     g[1] = 28
...     g[2] = 45
...     print("g =", g)
...
>>> f = [14, 23, 37]
>>> replace_contents(f)
g = [17, 28, 45]
>>> f
[17, 28, 45]
>>>
```

Function arguments are transferred using pass-by-object-reference.

References to objects are copied, not the objects themselves.

Return Semantics

```
>>> def f(d):
...     return d
...
>>> c = [6, 10, 16]
>>> e = f(c)
>>> c is e
True
>>>
```

Function Arguments

Default Argument Values

```
>>> def banner(message, border='-'):
...     line = border * len(message)
...     print(line)
...     print(message)
...     print(line)
...
>>> banner("Norwegian Blue")
--------------
Norwegian Blue
--------------
>>> banner("Sun, Moon and Stars", "*")
********************
Sun, Moon and Stars
********************
>>> banner("Sun, Moon and Stars", border="*")
********************
Sun, Moon and Stars
********************
>>> banner(border=".", message="Hello from Earth")
.................
Hello from Earth
.................
>>>
```

Arguments with default values must come after those without default values.

When Are Default Values Evaluated?

Default Value Evaluation

```
>>> import time
>>> time.ctime()
'Sun Nov 24 19:43:48 2019'
>>> def show_default(arg=time.ctime()):
...     print(arg)
...
>>> show_default()
Sun Nov 24 19:43:49 2019
>>> show_default()
Sun Nov 24 19:43:49 2019
>>> show_default()
Sun Nov 24 19:43:49 2019
>>>
```

Default Value Evaluation

Remember that `def` is a statement executed at runtime.

Default arguments are evaluated when `def` is executed.

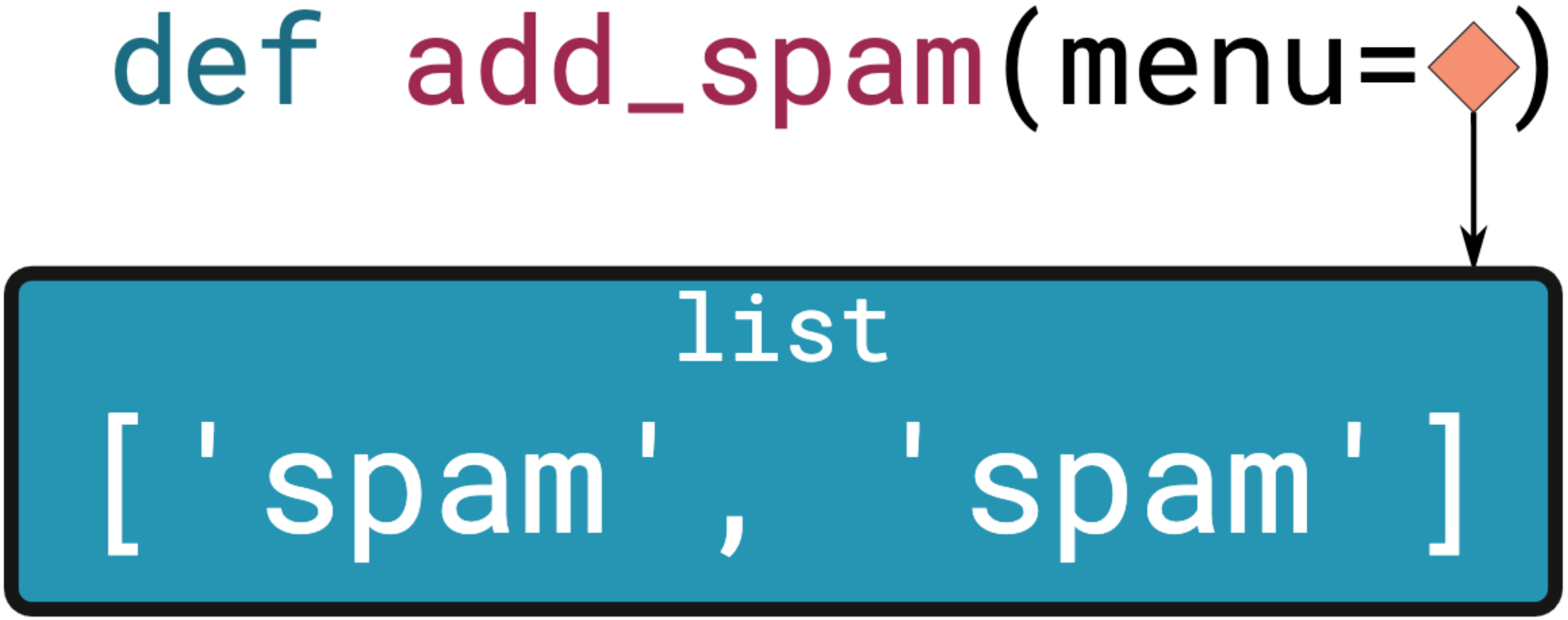
Immutable default values don't cause problems.

Mutable default values can cause confusing effects.

Mutable Default Values

```
>>> def add_spam(menu=[]):
...     menu.append("spam")
...     return menu
...
>>> breakfast = ['bacon', 'eggs']
>>> add_spam(breakfast)
['bacon', 'eggs', 'spam']
>>> lunch = ['baked beans']
>>> add_spam(lunch)
['baked beans', 'spam']
>>> add_spam()
['spam']
>>> add_spam()
['spam', 'spam']
>>> add_spam()
['spam', 'spam', 'spam']
>>> add_spam()
['spam', 'spam', 'spam', 'spam']
>>>
```

Modifying Mutable Default Values



Always use immutable objects for default values.

Immutable Default Value

```
>>> def add_spam(menu=None):
...     if menu is None:
...         menu = []
...     menu.append('spam')
...     return menu
...
>>> add_spam()
['spam']
>>> add_spam()
['spam']
>>> add_spam()
['spam']
>>>
```

Python's Type System

Python's Type System

```
>>> def add(a, b):
...     return a + b
...
>>> add(5, 7)
12
>>> add(3.1, 2.4)
5.5
>>> add("news", "paper")
'newspaper'
>>> add([1, 6], [21, 107])
[1, 6, 21, 107]
>>> add("The answer is", 42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in add
TypeError: can only concatenate str (not "int") to str
>>>
```

Python will not generally perform implicit conversions between types.

Type declarations are unnecessary in Python.

Names can be rebound as necessary to objects of any type.

Name resolution to objects is managed by scopes and scoping rules.

Scopes in Python

Local	Inside the current function
Enclosing	Inside enclosing functions
Global	At the top level of the module
Built-in	In the special `builtins` module

LEGB

Scopes in Python do no correspond to source code blocks.

Scopes

```
from urllib.request import urlopen
import sys

def fetch_words(url):
    story = urlopen(url)
    story_words = []
    for line in story:
        line_words = line.decode('utf8').split()
        for word in line_words:
            story_words.append(word)
    story.close()
    return story_words

def print_items(items):
    for item in items:
        print(item)

def main(url):
    words = fetch_words(url)
    print_items(words)

if __name__ == '__main__':
    main(sys.argv[1])
```

Rebinding Global Names

```
>>> count = 0
>>> def show_count():
...     print(count)
...
>>> def set_count(c):
...     count = c
...
>>> show_count()
0
>>> set_count(5)
>>> show_count()
0
>>> def set_count(c):
...     global count
...     count = c
...
>>> show_count()
0
>>> set_count(5)
>>> show_count()
5
>>>
```

`global`

Rebind global names into a local namespace.

Everything Is an Object

```
>>> import words
>>> type(words)
<class 'module'>
>>> dir(words)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'fetch_words', 'main', 'print_items', 'sys', 'urlopen
']
>>> type(words.fetch_words)
<class 'function'>
>>> dir(words.fetch_words)
['__annotations__', '__call__', '__class__', '__closure__', '__code__', '__defau
lts__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__get__', '__getattribute__', '__globals__', '__gt__', '__hash__', '
__init__', '__init_subclass__', '__kwdefaults__', '__le__', '__lt__', '__module_
_', '__name__', '__ne__', '__new__', '__qualname__', '__reduce__', '__reduce_ex_
_', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
>>> words.fetch_words.__name__
'fetch_words'
>>> words.fetch_words.__doc__
'Fetch a list of words from a URL.\n\n    Args:\n        url: The URL of a UTF-8
 text document.\n\n    Returns:\n        A list of strings containing the words
from\n        the document.\n    '
>>>
```

Summary

Python uses named references to objects

Assignment attaches a name to an object

Assigning one name to another makes them both point at the same object

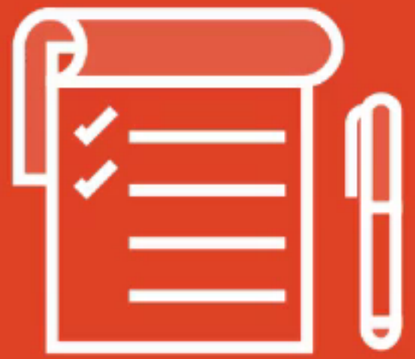
The garbage collector removes objects with no references

`id()` returns a unique integer ID for an object

`is` determines if two names refer to the same object

We can test for equivalence with ==

Summary



Function arguments are passed by object reference

Rebinding function arguments loses the original object reference

`return` passes back an object reference to the caller

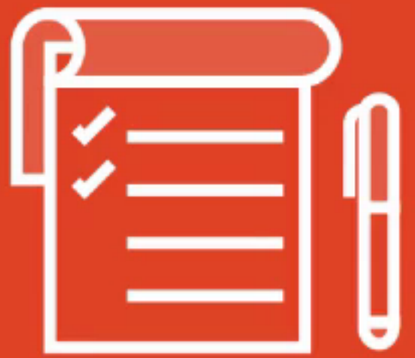
Function arguments may have a default value

Default argument values are evaluated once, when the function is defined

Python uses dynamic typing

Python has strong typing

Summary



Python names are looked up using the LEGB rule

Global references can be read from local scopes

Use `global` to assign to global references from a local scope

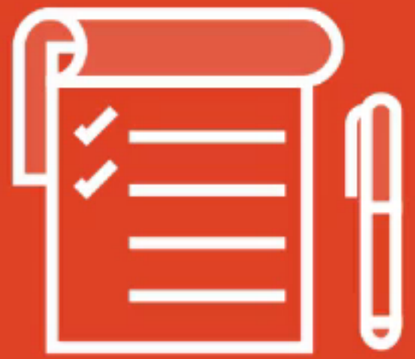
Everything in Python is an object

`import` and `def` bind names to objects

`type()` reports the type of an object

`dir()` introspects the attributes of an object

Summary



You can access the name of a function or module with `__name__`

Docstrings can be accessed through `__doc__`

You can use `len()` to measure the length of a string

The repetition operator, `*`, repeats a string an integral number of times