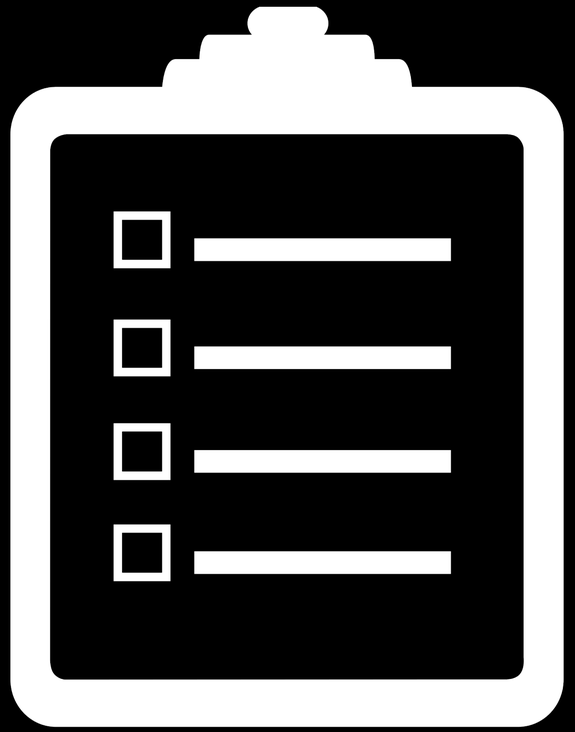
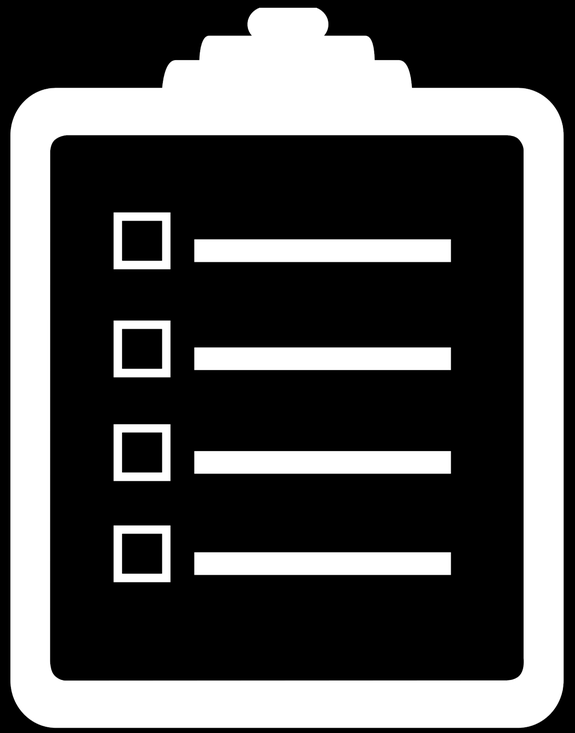
Python Fundamentals

April 6, 2017

Agenda



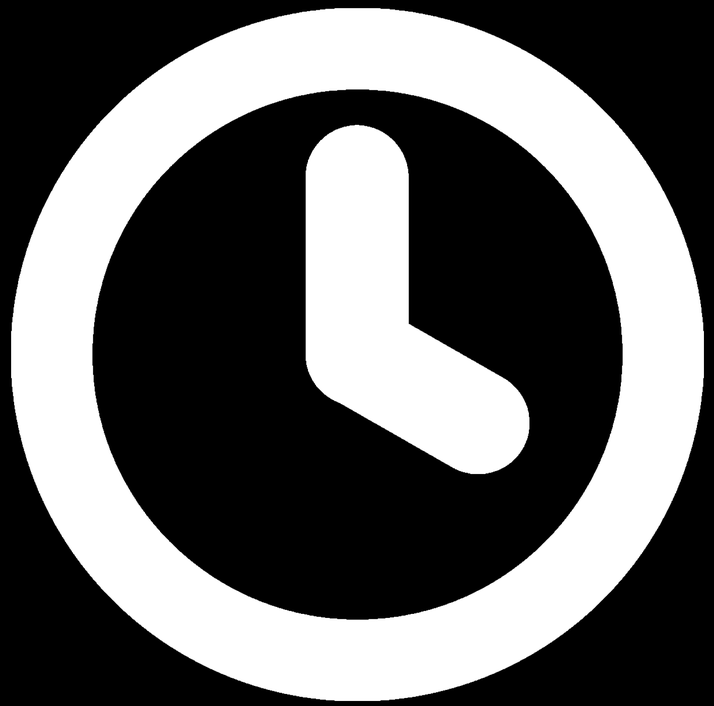
Agenda



- Brief Review
- Types Redux
- String Formatting
- File I/O
- Scripts, Modules, Imports
- Virtual Environments
- Lab!

From Last Time

Review



Zen of Python

Variables and Types

Numerics and Booleans

Strings and Slicing

Lists

Console I/O

Control Flow

Loops and range()

Zen of Python

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Variables

```
x = 2
x * 7
# => 14
```

No semicolon!

```
x = "Hello, I'm "
x + "Python!"
# => "Hello, I'm Python"
```

What happened here?!

Where's My Type?

Variables in Python are **dynamically-typed**: declared without an explicit type

However, **objects** have a type, so Python knows the type of a variable, even if you don't

```
type(1)          # => <class 'int'>
type("Hello")    # => <class 'str'>
type(None)       # => <class 'NoneType'>
```

This is the same object as the literal type `int`

```
type(int)        # => <class 'type'>
type(type)       # => <class 'type'>
```

Python's dynamic type system is fascinating

Numbers and Math

```
3                # => 3   (int)
3.0              # => 3.0 (float)


1 + 1            # => 2
8 - 1            # => 7
10 * 2           # => 20
9 / 3            # => 3.0
```

Python has two numeric types `int` and `float`

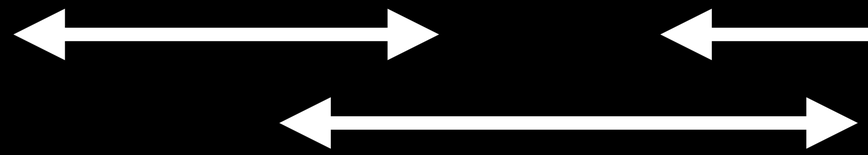
Booleans

```
True                # => True
False               # => False

not True            # => False
True and False      # => False
True or False       # => True (short-circuits)

2 * 3 == 5          # => False
2 * 3 != 5          # => True

1 < 10              # => True
1 < 2 < 3           # => True  (1 < 2 and 2 < 3)
```

`bool` is a subtype of `int`, where `True == 1` and `False == 0`

Strings

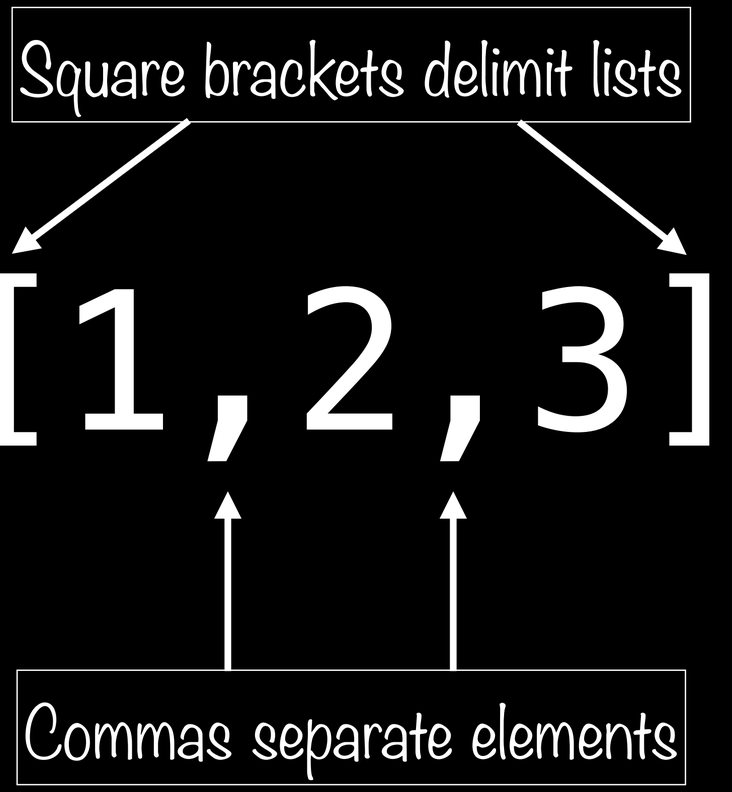
```
  0  1  2  3  4  5  6
s='Arthur'
 -6 -5 -4 -3 -2 -1  0
```



```
s[2] == 't'
s[-1] == 'r'
s[:2] == 'Ar'
s[1:-1] == 'rthu'
s[1:5:2] == 'rh'
s[::-1] == 'ruhtrA'
```

Lists

```
easy_as = [1,2,3]
```



Console I/O

`input` prompts the user for input

```
# Read a string from the user
>>> name = input("What is your name? ")
What is your name? Sam

>>> print("I'm Python. Nice to meet you,", name)
I'm Python. Nice to meet you, Sam
```

If Statements

No parentheses

Colon

No curly braces!

```
if the_world_is_flat:
    print("Don't fall off!")
```

Use 4 spaces to indent

New Stuff!

Truthy and Falsy

Truthy and Falsy

```
# 'Falsy'
bool(None)
bool(False)
bool(0)
bool(0.0)
bool('')
```

Truthy and Falsy

```
# 'Falsy'
bool(None)
bool(False)
bool(0)
bool(0.0)
bool('')

# Empty data structures are 'falsy'
bool([])  # => False
```

Truthy and Falsy

```
# 'Falsy'
bool(None)
bool(False)
bool(0)
bool(0.0)
bool('')

# Empty data structures are 'falsy'
bool([])  # => False

# Everything else is 'truthy'
```

Truthy and Falsy

```
# 'Falsy'
bool(None)
bool(False)
bool(0)
bool(0.0)
bool('')

# Empty data structures are 'falsy'
bool([])  # => False

# Everything else is 'truthy'

# How should we check for an empty list?
data = []
```

Truthy and Falsy

```
# 'Falsy'
bool(None)
bool(False)
bool(0)
bool(0.0)
bool('')

# Empty data structures are 'falsy'
bool([])  # => False

# Everything else is 'truthy'

# How should we check for an empty list?
data = []
if data:
    process(data):
```

Truthy and Falsy

```
# 'Falsy'
bool(None)
bool(False)
bool(0)
bool(0.0)
bool('')

# Empty data structures are 'falsy'
bool([])  # => False

# Everything else is 'truthy'

# How should we check for an empty list?
data = []
if data:
    process(data):
```

Notably, we don't use `if len(data) == 0`

Truthy and Falsy

```
# 'Falsy'
bool(None)
bool(False)
bool(0)
bool(0.0)
bool('')

# Empty data structures are 'falsy'
bool([])  # => False

# Everything else is 'truthy'

# How should we check for an empty list?
data = []
if data:
    process(data):
else:
    print("There's no data!")
```

Notably, we don't use `if len(data) == 0`

Truthy and Falsy

```
# 'Falsy'
bool(None)
bool(False)
bool(0)
bool(0.0)
bool('')

# Empty data structures are 'falsy'
bool([])  # => False

# Everything else is 'truthy'

# How should we check for an empty list?
data = []
if data:
    process(data):
else:
    print("There's no data!")
```

Notably, we don't use `if len(data) == 0`

You should almost never test `if expr == True`

Loops

For Loops

Loop explicitly over data

Strings, lists, etc.

```
for item in iterable:
    process(item)
```

No loop counter!

range

Used to iterate over
a sequence of numbers

range

Used to iterate over a sequence of numbers

```
range(3)
# generates 0, 1, 2
```

range

Used to iterate over a sequence of numbers

```
range(3)
# generates 0, 1, 2

range(5, 10)
# generates 5, 6, 7, 8, 9
```

range

Used to iterate over a sequence of numbers

```
range(3)
# generates 0, 1, 2

range(5, 10)
# generates 5, 6, 7, 8, 9

range(2, 12, 3)
# generates 2, 5, 8, 11
```

range

Used to iterate over a sequence of numbers

```
range(3)
# generates 0, 1, 2

range(5, 10)
# generates 5, 6, 7, 8, 9

range(2, 12, 3)
# generates 2, 5, 8, 11

range(-7, -30, -5)
# generates -7, -12, -17, -22, -27
```

range

Used to iterate over a sequence of numbers

```
range(3)
# generates 0, 1, 2

range(5, 10)
# generates 5, 6, 7, 8, 9

range(2, 12, 3)
# generates 2, 5, 8, 11

range(-7, -30, -5)
# generates -7, -12, -17, -22, -27
```

`range(stop)` or `range(start, stop[, step])`

`break` and `continue`

break and continue

```
for n in range(10):
    if n == 6:
        break
    print(n, end=',')
# => 0, 1, 2, 3, 4, 5,
```

`break` and `continue`

```
for n in range(10):
    if n == 6:
        break
    print(n, end=',')
# => 0, 1, 2, 3, 4, 5,
```

`break` breaks out of the smallest enclosing `for` or `while` loop

`break` and `continue`

```
for n in range(10):
    if n == 6:
        break
    print(n, end=',')
# => 0, 1, 2, 3, 4, 5,

for n in range(10):
    if n % 2 == 0:
        print("Even", n)
        continue
    print("Odd", n)
```

`break` breaks out of the smallest enclosing `for` or `while` loop

`break` and `continue`

```
for n in range(10):
    if n == 6:
        break
    print(n, end=',')
# => 0, 1, 2, 3, 4, 5,

for n in range(10):
    if n % 2 == 0:
        print("Even", n)
        continue
    print("Odd", n)
```

`break` breaks out of the smallest enclosing `for` or `while` loop

`continue` continues with the next iteration of the loop

Functions

Dive into Python functions Week 3

Writing Functions

The `def` keyword defines a function

Parameters have no explicit types

```
def fn_name(param1, param2):
    value = do_something()
    return value
```

`return` is optional
if either return or its value are omitted,
implicitly returns `None`

Prime Number Generator

Prime Number Generator

Prime Number Generator

```
def is_prime(n):
```

Prime Number Generator

```
def is_prime(n):
    for i in range(2, n):
```

Prime Number Generator

```
def is_prime(n):
    for i in range(2, n):
        if n % i == 0:
```

Prime Number Generator

```
def is_prime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
```

Prime Number Generator

```
def is_prime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

Prime Number Generator

```
def is_prime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

n = input("Enter a number: ")
```

Prime Number Generator

```
def is_prime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

n = input("Enter a number: ")
for x in range(2, int(n)):
```

Prime Number Generator

```
def is_prime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

n = input("Enter a number: ")
for x in range(2, int(n)):
    if is_prime(x):
        print(x, " is prime")
    else:
        print(x, " is not prime")
```

More on Functions Later (Week 3)

Default Argument Values

Keyword Arguments

Variadic Argument Lists

Unpacking Arguments

Anonymous Functions

First-Class Functions

Functional Programming

Objects and Types Redux

Objects are typed
Variables are untyped

Objects

Everything is an Object

Everything is an Object

```
isinstance(4, object)          # => True
```

Everything is an Object

```
isinstance(4, object)          # => True

isinstance("Hello", object)    # => True
```

Everything is an Object

```
isinstance(4, object)          # => True

isinstance("Hello", object)    # => True

isinstance(None, object)       # => True
```

Everything is an Object

```
isinstance(4, object)          # => True

isinstance("Hello", object)    # => True

isinstance(None, object)       # => True

isinstance([1,2,3], object)    # => True
```

Objects have Identity

Objects have Identity

`id(object)` gives object's "identity"

Objects have Identity

`id(object)` gives object's "identity"

"Identity" is unique and fixed during an object's lifetime

Objects have Identity

`id(object)` gives object's "identity"

"Identity" is unique and fixed during an object's lifetime

Objects are tagged with their type at runtime

Objects have Identity

`id(object)` gives object's "identity"

"Identity" is unique and fixed during an object's lifetime

Objects are tagged with their type at runtime

Objects contain pointers to their data blob

Objects have Identity

`id(object)` gives object's "identity"

"Identity" is unique and fixed during an object's lifetime

Objects are tagged with their type at runtime

Objects contain pointers to their data blob

This means even small things take up a lot of space!

Objects have Identity

`id(object)` gives object's "identity"

"Identity" is unique and fixed during an object's lifetime

Objects are tagged with their type at runtime

Objects contain pointers to their data blob

This means even small things take up a lot of space!

```
(4).__sizeof__()
 # => 28 (bytes)
```

Objects have Identity

`id(object)` gives object's "identity"

"Identity" is unique and fixed during an object's lifetime

Objects are tagged with their type at runtime

Objects contain pointers to their data blob

This means even small things take up a lot of space!

```
(4).__sizeof__()
 # => 28 (bytes)
```

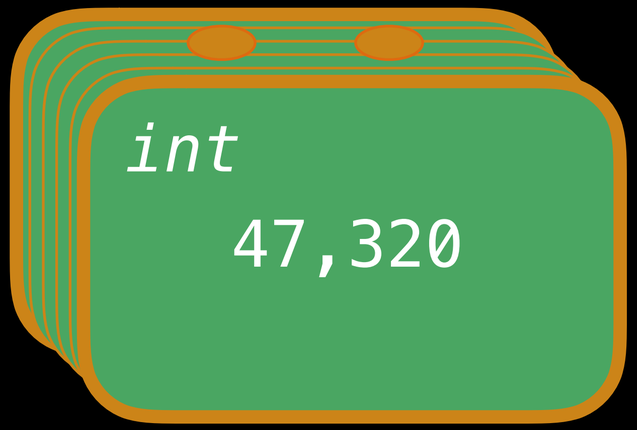
In CPython (reference implementation in C),
`id` gives the PyObject's memory address

An Analogy

An Analogy

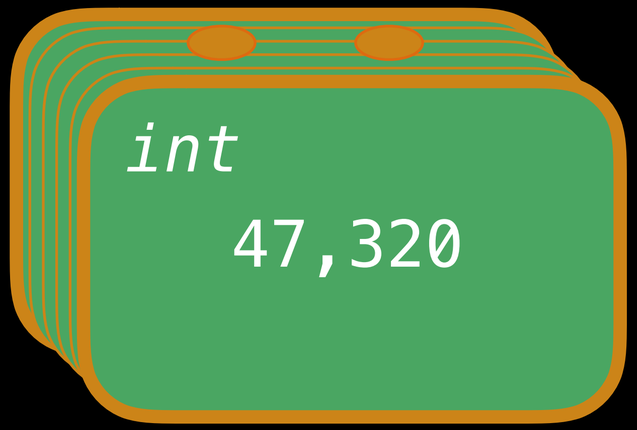
Imagine a Python object as a suitcase.
They come in different sizes and store different values.
Each suitcase has a given type, and holds some value

An Analogy



Imagine a Python object as a suitcase.
They come in different sizes and store different values.
Each suitcase has a given type, and holds some value

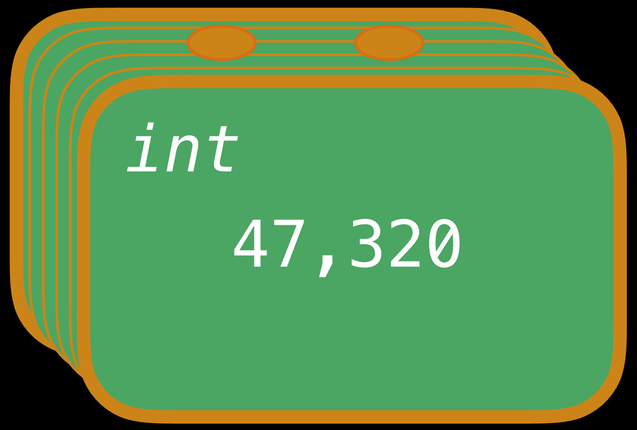
An Analogy



Imagine a Python object as a suitcase.
They come in different sizes and store different values.
Each suitcase has a given type, and holds some value

Python handles all of your baggage (objects) for you!

An Analogy



Imagine a Python object as a suitcase.
They come in different sizes and store different values.
Each suitcase has a given type, and holds some value

Python handles all of your baggage (objects) for you!

Variables

Variables

Variables

Variables are references to objects

Variables

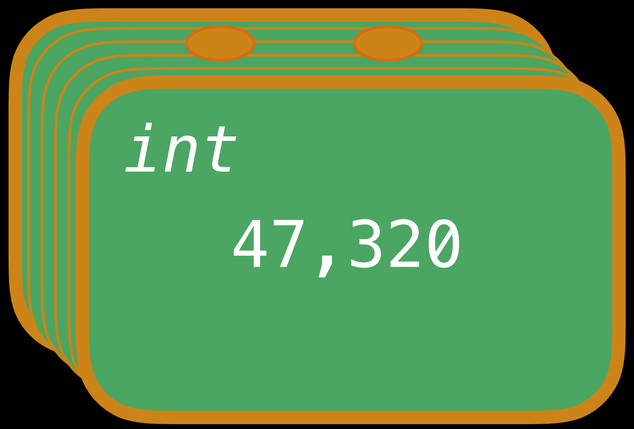
Variables are references to objects

Little more than a pointer

Variables

Variables are references to objects

Little more than a pointer

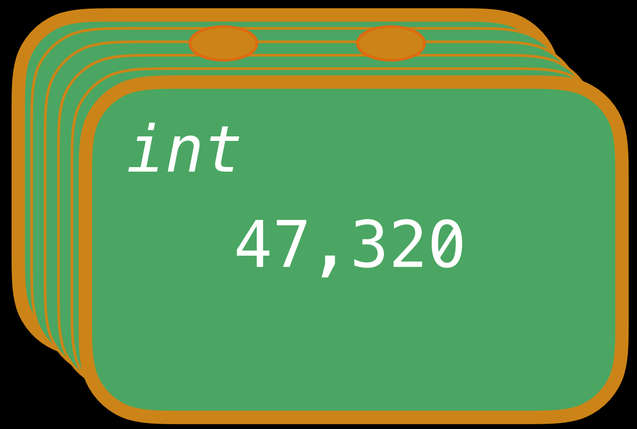


Variables

Variables are references to objects

Little more than a pointer

```
x = 47320
```



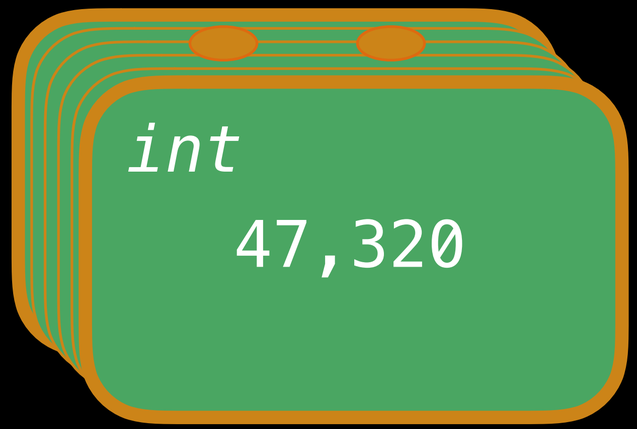
Variables

Variables are references to objects

Little more than a pointer

In our analogy, a variable is a label for your baggage

```
x = 47320
```



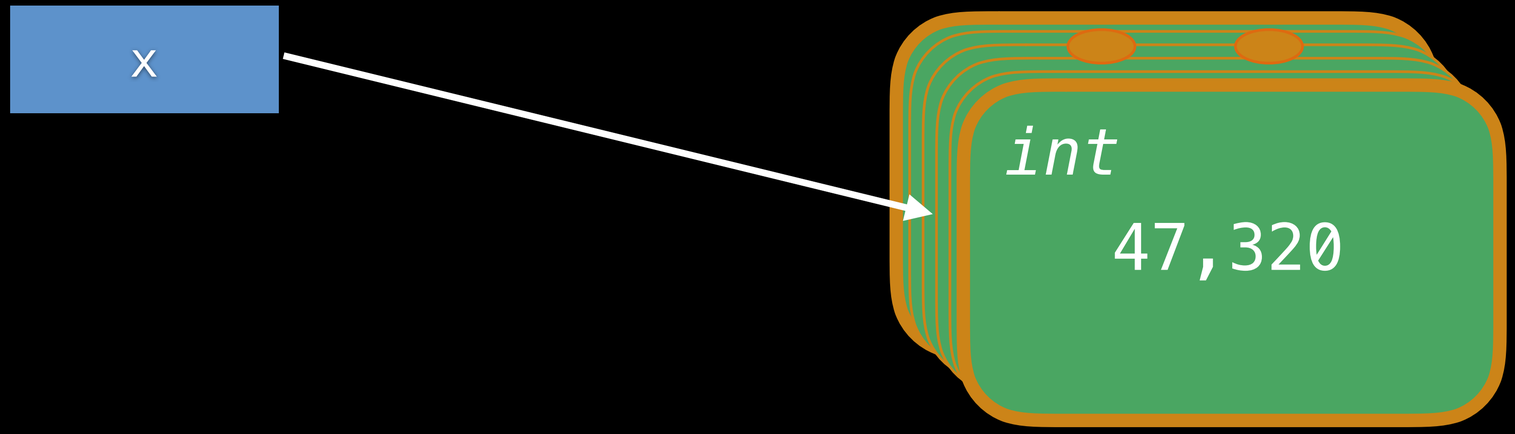
Variables

Variables are references to objects

Little more than a pointer

In our analogy, a variable is a label for your baggage

```
x = 47320
```



Symbol Table

Symbol Table

Remember, "Namespaces are one honking great idea!"

Symbol Table

Remember, "Namespaces are one honking great idea!"

A Python namespace maintains information about labels

Symbol Table

Remember, "Namespaces are one honking great idea!"

A Python namespace maintains information about labels

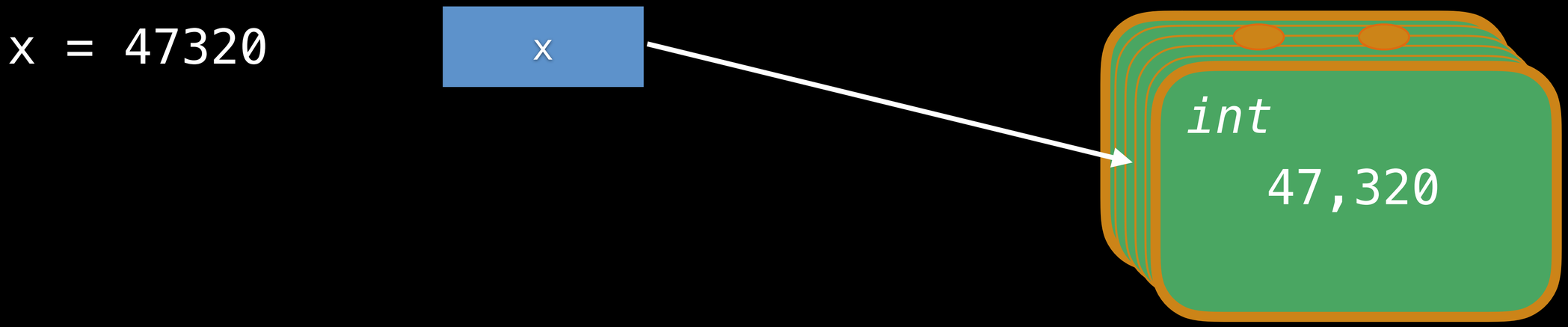
Local namespace (via `locals()`), global, module, and more!

Symbol Table

Remember, "Namespaces are one honking great idea!"

A Python namespace maintains information about labels

Local namespace (via `locals()`), global, module, and more!

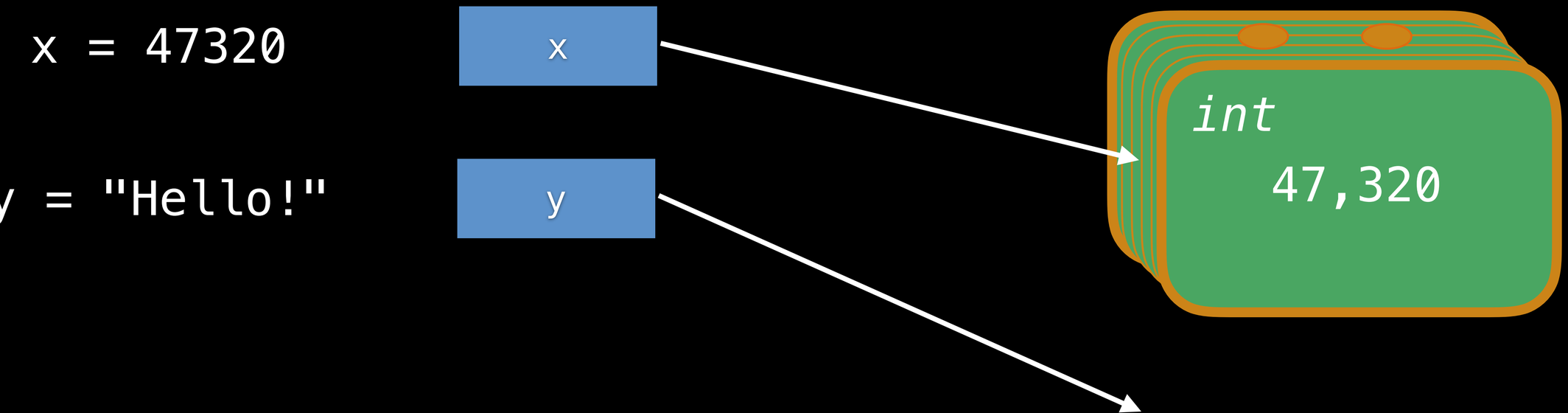


Symbol Table

Remember, "Namespaces are one honking great idea!"

A Python namespace maintains information about labels

Local namespace (via `locals()`), global, module, and more!

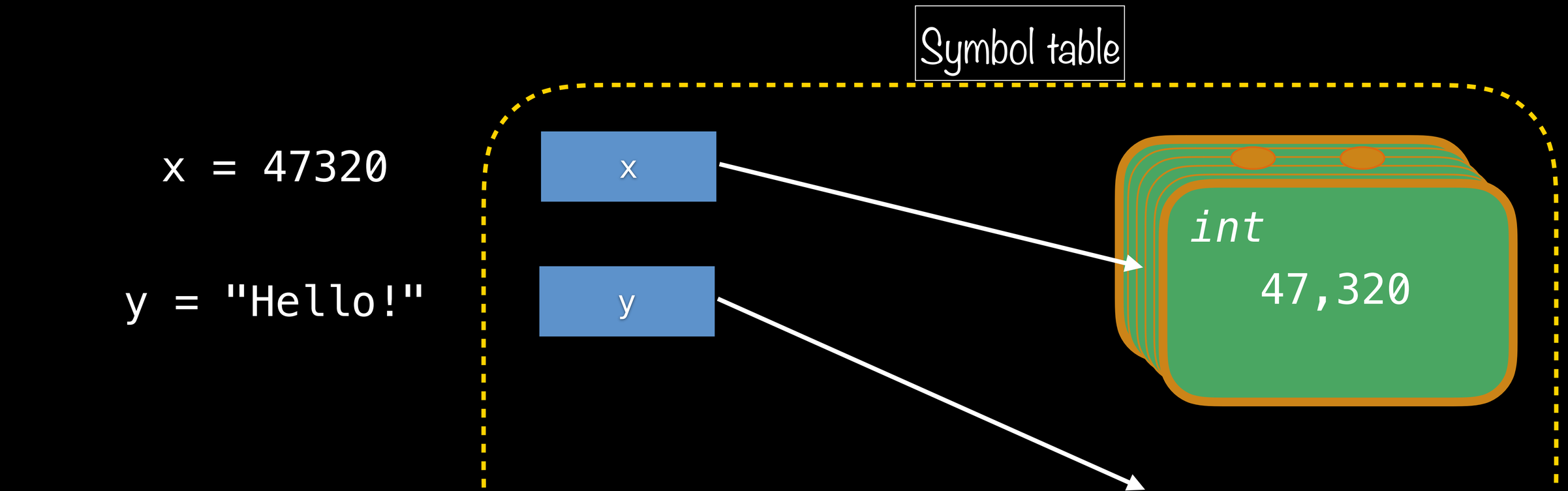


Symbol Table

Remember, "Namespaces are one honking great idea!"

A Python namespace maintains information about labels

Local namespace (via `locals()`), global, module, and more!



Variable Assignment

Variable Assignment

Assigning from a variable does **not** copy an object.

Variable Assignment

Assigning from a variable does **not** copy an object.

Instead, it adds another reference to the same object.

Variable Assignment

Assigning from a variable does **not** copy an object.

Instead, it adds another reference to the same object.

Python will **always** handle the creation of new objects.

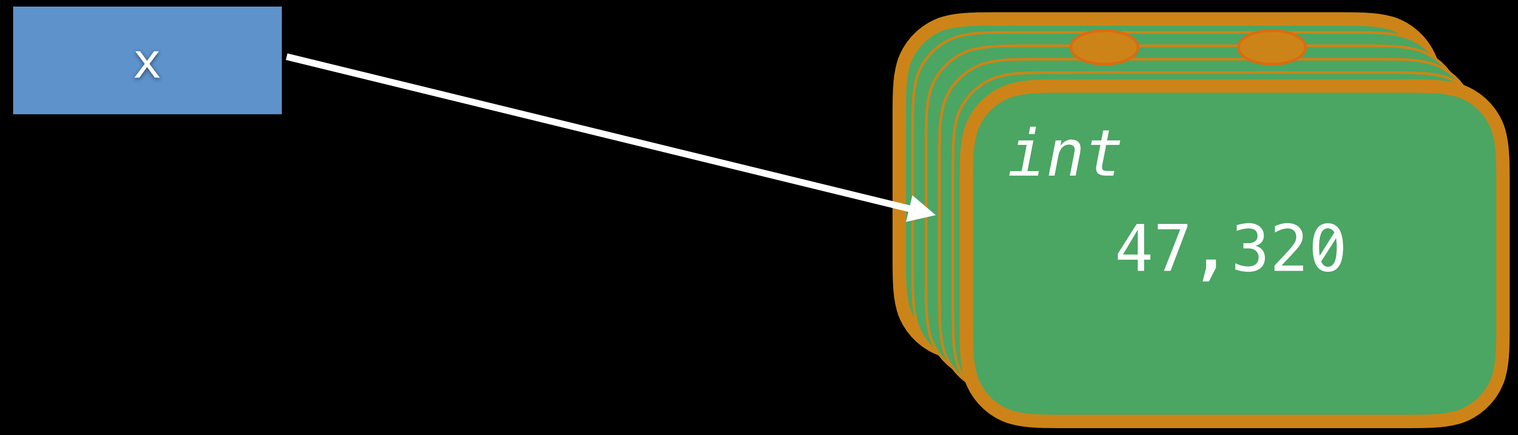
Variable Assignment

Assigning from a variable does **not** copy an object.

Instead, it adds another reference to the same object.

Python will **always** handle the creation of new objects.

```
x = 47320
```



Variable Assignment

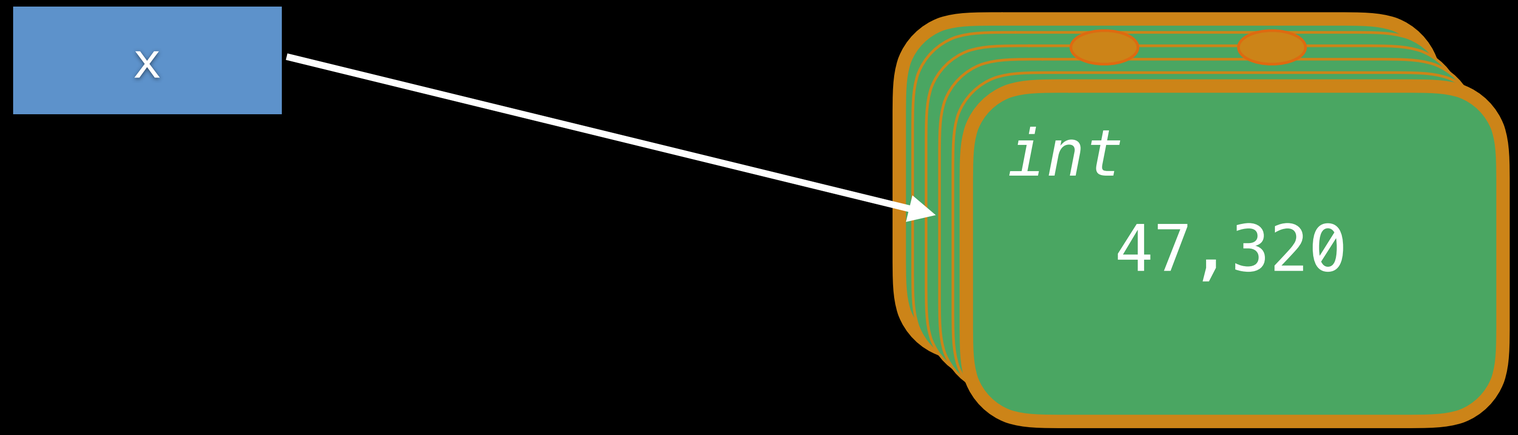
Assigning from a variable does **not** copy an object.

Instead, it adds another reference to the same object.

Python will **always** handle the creation of new objects.

```
x = 47320
```

```
y = x
```



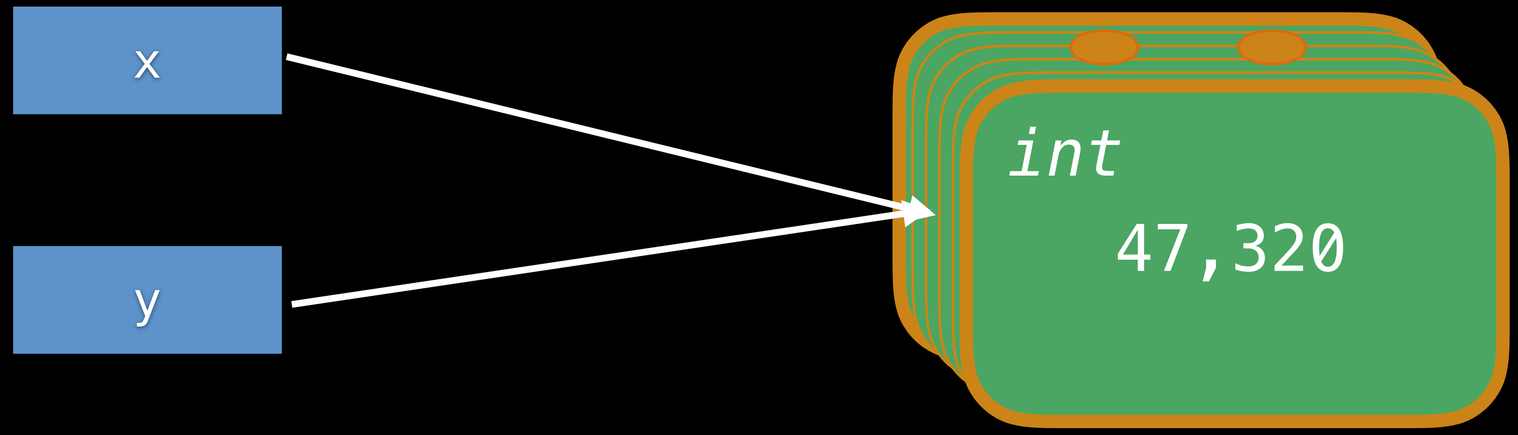
Variable Assignment

Assigning from a variable does **not** copy an object.

Instead, it adds another reference to the same object.

Python will **always** handle the creation of new objects.

```
x = 47320

y = x
```



Duck Typing

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

James Whitcomb Riley

Duck Typing

Duck Typing

```
def compute(a, b, c):
    return (a + b) * c
```

Duck Typing

```
def compute(a, b, c):
    return (a + b) * c

compute(1, 2, 3)  # => 9
```

Duck Typing

```
def compute(a, b, c):
    return (a + b) * c

compute(1, 2, 3)  # => 9
compute([1], [2, 3], 2) # => [1, 2, 3, 1, 2, 3]
```

Duck Typing

```
def compute(a, b, c):
    return (a + b) * c

compute(1, 2, 3)  # => 9
compute([1], [2, 3], 2) # => [1, 2, 3, 1, 2, 3]
compute('l', 'olo', 4)  # => 'lolololololololololo'
```

Duck Typing

```
def compute(a, b, c):
    return (a + b) * c

compute(1, 2, 3)  # => 9
compute([1], [2, 3], 2) # => [1, 2, 3, 1, 2, 3]
compute('l', 'olo', 4)  # => 'lolololololololololo'
```

For `compute`, all that matters is that the arguments support `+` and `*`

Duck Typing

Duck Typing

If you can `walk`, `swim`, and `quack`, then you're a `Duck`

Duck Typing

If you can `walk`, `swim`, and `quack`, then you're a `Duck`

Promotes interface-style generic programming

Duck Typing

If you can `walk`, `swim`, and `quack`, then you're a `Duck`

Promotes interface-style generic programming

We'll see more later - stay tuned!

Aside: `is` vs. `==`

is vs ==

`is` vs `==`

We've seen `==` for equality testing

```
1 == 1.0
```

`is` vs `==`

We've seen `==` for equality testing

```
1 == 1.0
```

True!

but we know these are different in some fundamental way

```
type(1) != type(1.0)
```

```
int != float
```

`is` vs `==`

We've seen `==` for equality testing

```
1 == 1.0
```

True!

but we know these are different in some fundamental way

```
type(1) != type(1.0)
```

```
int != float
```

`a is not b`

is syntactic sugar for

`not (a is b)`

The `is` operator checks *identity* instead of *equality*

`is` vs `==`

We've seen `==` for equality testing

```
1 == 1.0
```

True!

but we know these are different in some fundamental way

```
type(1) != type(1.0)
```

```
int != float
```

`a is not b`
is syntactic sugar for
`not (a is b)`

The `is` operator checks *identity* instead of *equality*

When comparing against **`None`** or other singletons,
always use **`is None`** instead of **`== None`**

Analogy

`is` checks if the suitcases are the same

`==` checks if they have the same stuff inside

Use `==` when comparing values

Use `is` when comparing identities

String Redux

Special Characters

Special Characters

```
print('doesn\'t')  # => doesn't
print("doesn't")   # => doesn't
```

Special Characters

```
print('doesn\'t')  # => doesn't
print("doesn't")   # => doesn't

print('"Yes," he said.')    # => "Yes," he said.
print("\"Yes,\" he said.")  # => "Yes," he said.
```

Special Characters

```
print('doesn\'t')  # => doesn't
print("doesn't")   # => doesn't

print('"Yes," he said.')    # => "Yes," he said.
print("\"Yes,\" he said.")  # => "Yes," he said.

print('"Isn\'t," she said.')  # => "Isn't," she said.
```

Special Characters

```
print('doesn\'t')  # => doesn't
print("doesn't")   # => doesn't

print('"Yes," he said.')    # => "Yes," he said.
print("\"Yes,\" he said.")  # => "Yes," he said.

print('"Isn\'t," she said.')  # => "Isn't," she said.
```

Choose the easiest string delimiter to work with!

Useful String Methods

Useful String Methods

```
greeting = "Hello world! "
```

Useful String Methods

```
greeting = "Hello world! "

greeting[4]             # => 'o'
'world' in greeting     # => True
len(greeting)           # => 13
```

Useful String Methods

```
greeting = "Hello world! "

greeting[4]           # => 'o'
'world' in greeting   # => True
len(greeting)         # => 13

greeting.find('lo')           # => 3 (-1 if not found)
greeting.replace('llo', 'y') # => "hey world!"
greeting.startswith('hell')   # => True
greeting.isalpha()            # => False (due to '!')
```

Useful String Methods

Useful String Methods

```
greeting = "Hello world! "
```

Useful String Methods

```
greeting = "Hello world! "

greeting.lower()  # => "hello world! "
greeting.title()  # => "Hello World! "
```

Useful String Methods

```
greeting = "Hello world! "

greeting.lower()  # => "hello world! "
greeting.title()  # => "Hello World! "

greeting.strip()  # => "Hello world!"
greeting.strip('! ')  # => "Hello world" (no '!')
```

Strings <—> Lists

Strings <—> Lists

```
# `split` partitions a string by a delimiter
```

Strings <—> Lists

```
# `split` partitions a string by a delimiter
'ham cheese bacon'.split()
# => ['ham', 'cheese', 'bacon']
```

Strings <—> Lists

```
# `split` partitions a string by a delimiter
'ham cheese bacon'.split()
# => ['ham', 'cheese', 'bacon']
'03-30-2016'.split(sep='-')
# => ['03', '30', '2016']
```

Strings <—> Lists

```
# `split` partitions a string by a delimiter
'ham cheese bacon'.split()
# => ['ham', 'cheese', 'bacon']
'03-30-2016'.split(sep='-')
# => ['03', '30', '2016']

# `join` creates a string from a list
```

Strings <—> Lists

```
# `split` partitions a string by a delimiter
'ham cheese bacon'.split()
# => ['ham', 'cheese', 'bacon']
'03-30-2016'.split(sep='-')
# => ['03', '30', '2016']

# `join` creates a string from a list
', '.join(['Eric', 'John', 'Michael'])
# => "Eric, John, Michael"
```

String Formatting 2.0!

String Formatting 2.0!

```
# Curly braces in strings are placeholders
'{} {}'.format('monty', 'python')  # => 'monty python'
```

String Formatting 2.0!

```
# Curly braces in strings are placeholders
'{} {}'.format('monty', 'python')  # => 'monty python'

# Provide values by position or by placeholder
"{0} can be {1} {0}s".format("strings", "formatted")
"{name} loves {food}".format(name="Sam", food="plums")
```

String Formatting 2.0!

```
# Curly braces in strings are placeholders
'{} {}'.format('monty', 'python')  # => 'monty python'

# Provide values by position or by placeholder
"{0} can be {1} {0}s".format("strings", "formatted")
"{name} loves {food}".format(name="Sam", food="plums")

# Pro: Values are converted to strings
"{} squared is {}".format(5, 5 ** 2)
```

Fancier Formatting

Fancier Formatting

```
# You can use C-style specifiers too!
"{:06.2f}".format(3.14159)  # => 003.14
```

Fancier Formatting

```
# You can use C-style specifiers too!
"{:06.2f}".format(3.14159)  # => 003.14

# Padding is just another specifier
'{:10}'.format('hi')  # => 'hi        '
'{:^12}'.format('TEST')  # => '****TEST****'
```

Fancier Formatting

```
# You can use C-style specifiers too!
"{:06.2f}".format(3.14159)  # => 003.14

# Padding is just another specifier
'{:10}'.format('hi')  # => 'hi        '
'{:^12}'.format('TEST')  # => '****TEST****'

# You can even look up values!
captains = ['Kirk', 'Picard']
"{caps[0]} > {caps[1]}".format(caps=captains)
```

Fancier Formatting

```
# You can use C-style specifiers too!
"{:06.2f}".format(3.14159)  # => 003.14

# Padding is just another specifier
'{:10}'.format('hi')  # => 'hi        '
'{:^12}'.format('TEST')  # => '****TEST****'

# You can even look up values!
captains = ['Kirk', 'Picard']
"{caps[0]} > {caps[1]}".format(caps=captains)
```

More techniques online!

Aside: Other Formatting Approaches

Aside: Other Formatting Approaches

```
# Pure C-style formatting
"%s, %s, %s. (Act %d)" % ('Words', 'words', 'words', 2)
#=> Words, words, words. (Act 2)
```

Aside: Other Formatting Approaches

```
# Pure C-style formatting
"%s, %s, %s. (Act %d)" % ('Words', 'words', 'words', 2)
#=> Words, words, words. (Act 2)
```

Problem: Not readable, value duplication, tuple unpacking

Aside: Other Formatting Approaches

```
# Pure C-style formatting
"%s, %s, %s. (Act %d)" % ('Words', 'words', 'words', 2)
#=> Words, words, words. (Act 2)
```

Problem: Not readable, value duplication, tuple unpacking

```
# String concatenation with +
"I am " + str(age) + " years old."
```

Aside: Other Formatting Approaches

```
# Pure C-style formatting
"%s, %s, %s. (Act %d)" % ('Words', 'words', 'words', 2)
#=> Words, words, words. (Act 2)
```

Problem: Not readable, value duplication, tuple unpacking

```
# String concatenation with +
"I am " + str(age) + " years old."
```

Problem: Slow, requires explicit string conversion

Aside: Other Formatting Approaches

```
# Pure C-style formatting
"%s, %s, %s. (Act %d)" % ('Words', 'words', 'words', 2)
#=> Words, words, words. (Act 2)
```

Problem: Not readable, value duplication, tuple unpacking

```
# String concatenation with +
"I am " + str(age) + " years old."
```

Problem: Slow, requires explicit string conversion

Using `.format()` is the fastest and most convenient

Time-Out for Announcements

Announcements

Announcements

CS41 on Piazza Course announcements, Q&A, Enroll!

Announcements

CS41 on Piazza Course announcements, Q&A, Enroll!

Auditors Keep an eye out for an incoming email.

Announcements

CS41 on Piazza Course announcements, Q&A, Enroll!

Auditors Keep an eye out for an incoming email.

Axess Enrollment if and only if received email saying so.

Announcements

CS41 on Piazza Course announcements, Q&A, Enroll!

Auditors Keep an eye out for an incoming email.

Axess Enrollment if and only if received email saying so.

Lecture Recordings Slides always, hopefully videos.

Announcements

CS41 on Piazza Course announcements, Q&A, Enroll!

Auditors Keep an eye out for an incoming email.

Axess Enrollment if and only if received email saying so.

Lecture Recordings Slides always, hopefully videos.

Assignment 0 Warmup to installation and submission.

Back to Python!

File I/O

File I/O

File Object (**f**)

Python Data

```
f = open(filename, mode)
```

File I/O

File Object (**f**)

Python Data

File I/O

```
f = open(filename, mode)
```

Relative or Absolute

File Object (**f**)

Python Data

File I/O

```
f = open(filename, mode)
```

Relative or Absolute

r read
w write
b binary

File Object (**f**)

Python Data

File I/O

```
f = open(filename, mode)
```

Relative or Absolute

r read
w write
b binary

File Object (f)

Python Data

File I/O

```
f = open(filename, mode)
```

Relative or Absolute

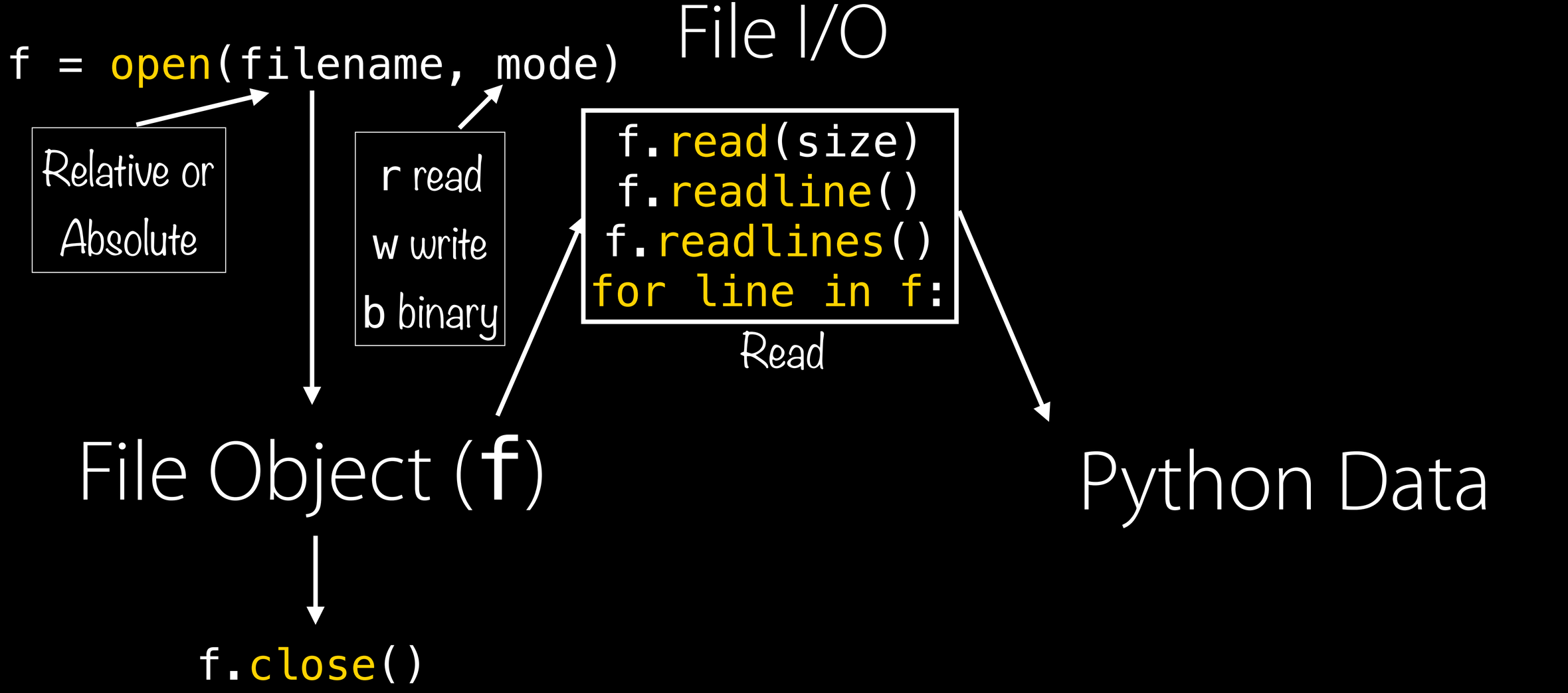
r read
w write
b binary

File Object (**f**)

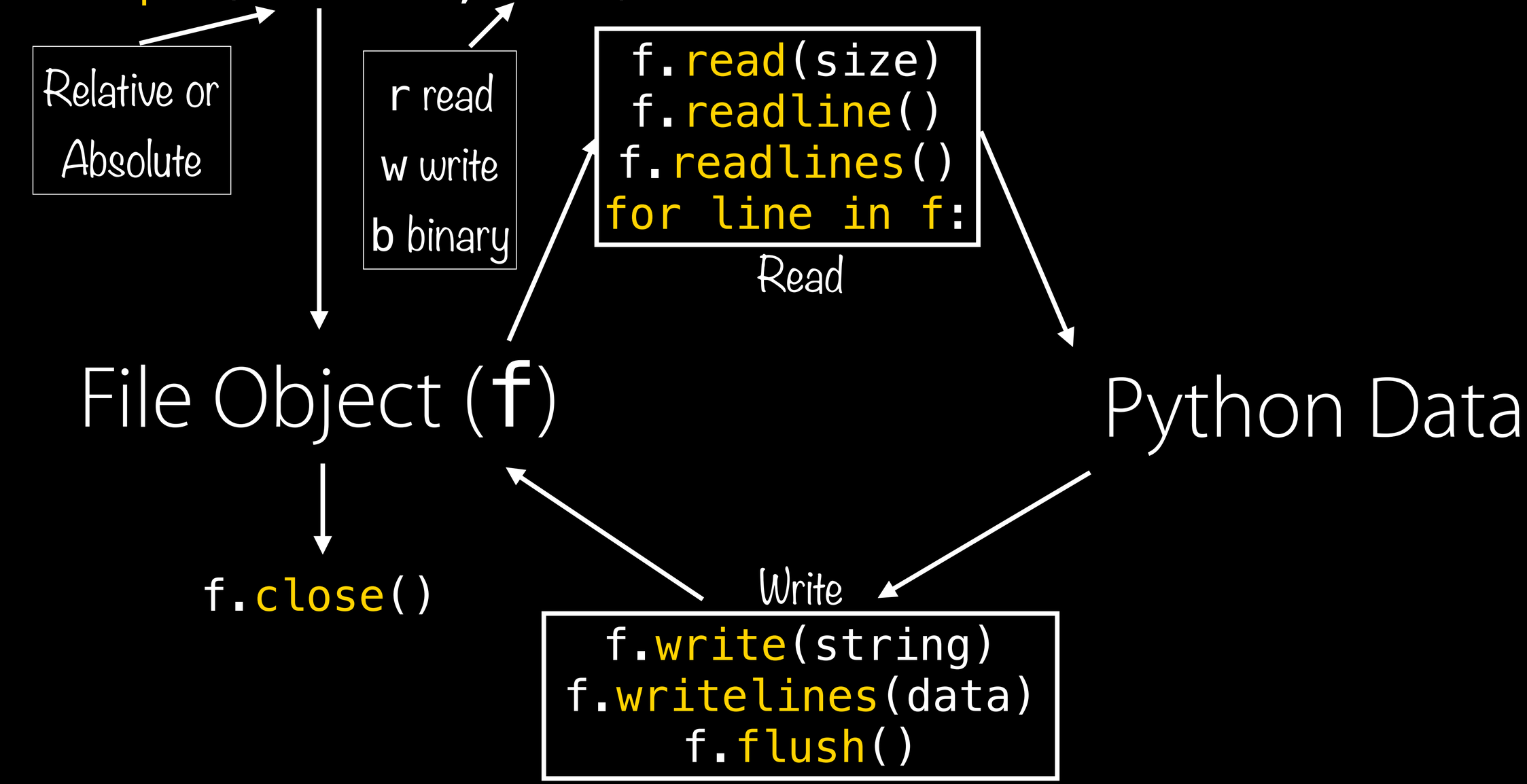
```
f.close()
```

Python Data

File I/O


f = open(filename, mode)
Relative or
Absolute
r read
w write
b binary
f.read(size)
f.readline()
f.readlines()
for line in f:
Read
File Object (f)
Python Data
f.close()

File I/O


f = open(filename, mode)
Relative or Absolute
r read
w write
b binary
f.read(size)
f.readline()
f.readlines()
for line in f:
Read
File Object (f)
Python Data
f.close()
Write
f.write(string)
f.writelines(data)
f.flush()

What if…?

```
f = open('file.txt', 'r')
print(1 / 0)  # Crash!
f.close()
```

What if…?

```
f = open('file.txt', 'r')
print(1 / 0)  # Crash!
f.close()
```

The file is never closed!

Be Responsible!

Be Responsible!

```
with open('file.txt', 'r') as f:
    content = f.read()
    print(1/0)
```

Be Responsible!

```
with open('file.txt', 'r') as f:
    content = f.read()
    print(1/0)
```

The `with expr as var` construct ensures that `expr` will be entered and exited regardless of the code block execution

Be Responsible!

```
with open('file.txt', 'r') as f:
    content = f.read()
    print(1/0)


f.closed  # => True
```

The `with expr as var` construct ensures that `expr` will be entered and exited regardless of the code block execution

Be Responsible!

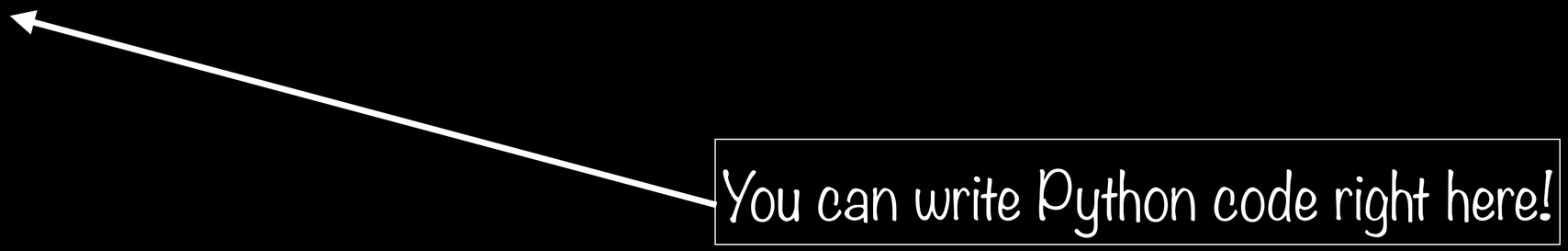
```
with open('file.txt', 'r') as f:
    content = f.read()
    print(1/0)


f.closed  # => True

""" `content` is still in scope """
'content' in locals()  # => True
```

The `with expr as var` construct ensures that `expr` will be entered and exited regardless of the code block execution

Scripts, Modules, Imports

Recall: Interactive Interpreter

```
sredmond$ python3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```



Interactive Interpreter

Interactive Interpreter

Problem Temporary

Interactive Interpreter

Problem Temporary

Solution Write code in a file!

First Script

First Script

```
#!/usr/bin/env python3 -tt
""" File: hello.py """
```

Shebang line specifies default executable and options

hello.py

First Script

```
#!/usr/bin/env python3 -tt
""" File: hello.py """

def greet(name):
    print("Hey {}, I'm Python!".format(name))
```

Shebang line specifies default executable and options

hello.py

First Script

```
#!/usr/bin/env python3 -tt
""" File: hello.py """

def greet(name):
    print("Hey {}, I'm Python!".format(name))

# Run only if called as a script
if __name__ == '__main__':
    name = input("What is your name? ")
    greet(name)
```

Shebang line specifies default executable and options

The special `__name__` variable is set to `'__main__'` if your file is executed as a script

hello.py

Running Python Scripts

We supply the filename of the Python script to run after the `python3` command

Running Python Scripts

```
sredmond$ python3 filename.py
```

We supply the filename of the Python script to run after the `python3` command

Running Python Scripts

```
sredmond$ python3 filename.py
<output from the file>
```

We supply the filename of the Python script to run after the `python3` command

Running Python Scripts

```
sredmond$ python3 filename.py
<output from the file>

sredmond$ python3 hello.py
```

We supply the filename of the Python script to run after the `python3` command

Running Python Scripts

```
sredmond$ python3 filename.py
<output from the file>
```

We supply the filename of the Python script to run after the `python3` command

```
sredmond$ python3 hello.py
What is your name? Sam
```

Running Python Scripts

```
sredmond$ python3 filename.py
<output from the file>
```

We supply the filename of the Python script to run after the `python3` command

```
sredmond$ python3 hello.py
What is your name? Sam
Hey Sam, I'm Python!
sredmond$
```

Python Scripts

Supplying the `-i` option (for 'interactive') will enter the interactive interpreter after running the python script

Python Scripts

```
sredmond$ python3 -i filename.py
<output from the file>
>>>
```

Supplying the `-i` option (for 'interactive') will enter the interactive interpreter after running the python script

Python Scripts

```
sredmond$ python3 -i filename.py
<output from the file>
>>>
```

Supplying the `-i` option (for 'interactive') will enter the interactive interpreter after running the python script

Now we have access to symbols from our script. Great for debugging!

Interactive Scripts

Interactive Scripts

```
sredmond$ python3 -i hello.py
```

Interactive Scripts

```
sredmond$ python3 -i hello.py
What is your name? Sam
Hey Sam, I'm Python!
```

Interactive Scripts

```
sredmond$ python3 -i hello.py
What is your name? Sam
Hey Sam, I'm Python!
>>> greet("Brexton")
Hey Brexton, I'm Python!
>>>
```

Executable Scripts

Executable Scripts

```
sredmond$ chmod +x hello.py
```

Executable Scripts

```
sredmond$ chmod +x hello.py
sredmond$ ./hello.py
```

Executable Scripts

```
sredmond$ chmod +x hello.py
sredmond$ ./hello.py
What is your name? Sam
```

Executable Scripts

```
sredmond$ chmod +x hello.py
sredmond$ ./hello.py
What is your name? Sam
Hey Sam, I'm Python!
sredmond$
```

Imports

Using Modules

Using Modules

```
# Import a module
import math
math.sqrt(16)  # => 4
```

Using Modules

```
# Import a module
import math
math.sqrt(16)  # => 4

# Import specific symbols from a module into the local namespace
from math import ceil, floor
ceil(3.7)  # => 4.0
floor(3.7)   # => 3.0
```

Using Modules

```
# Import a module
import math
math.sqrt(16)  # => 4

# Import specific symbols from a module into the local namespace
from math import ceil, floor
ceil(3.7)  # => 4.0
floor(3.7)   # => 3.0

# Bind module symbols to a new symbol in the local namespace
from some_module import super_long_symbol_name as short_name
```

Using Modules

```
# Import a module
import math
math.sqrt(16)  # => 4

# Import specific symbols from a module into the local namespace
from math import ceil, floor
ceil(3.7)  # => 4.0
floor(3.7)   # => 3.0

# Bind module symbols to a new symbol in the local namespace
from some_module import super_long_symbol_name as short_name

# Any python file (including those you write) is a module
from my_script import my_function, my_variable
```

Using Modules

```
# Import a module
import math
math.sqrt(16)  # => 4

# Import specific symbols from a module into the local namespace
from math import ceil, floor
ceil(3.7)  # => 4.0
floor(3.7)   # => 3.0

# Bind module symbols to a new symbol in the local namespace
from some_module import super_long_symbol_name as short_name

# Any python file (including those you write) is a module
from my_script import my_function, my_variable
```

We almost always import the whole module, rather than specific symbols

Virtual Environments

Virtual Environments: Problem

Many computers come with Python 2 installed.

You may have installed libraries for other classes.

We want an isolated Python environment for this class.

An Analogy

An Analogy

The Beach

An Analogy

An Analogy


The Beach
Default Tools
Old Bucket
Broken Shovel
Cracked Rake

An Analogy

The Beach

Default Tools
Old Bucket
Broken Shovel
Cracked Rake

I want to build a sand castle! But I don't want to use the default tools.

An Analogy

The Beach

My Sand Castle

Default Tools
Old Bucket
Broken Shovel
Cracked Rake

I want to build a sand castle! But I don't want to use the default tools.

An Analogy

The Beach

Default Tools
Old Bucket
Broken Shovel
Cracked Rake

My Sand Castle
Bucket? Old

I want to build a sand castle! But I don't want to use the default tools.

An Analogy

The Beach

Default Tools
Old Bucket
Broken Shovel
Cracked Rake

My Sand Castle
Bucket? Old
Shovel? Broken

I want to build a sand castle! But I don't want to use the default tools.

An Analogy

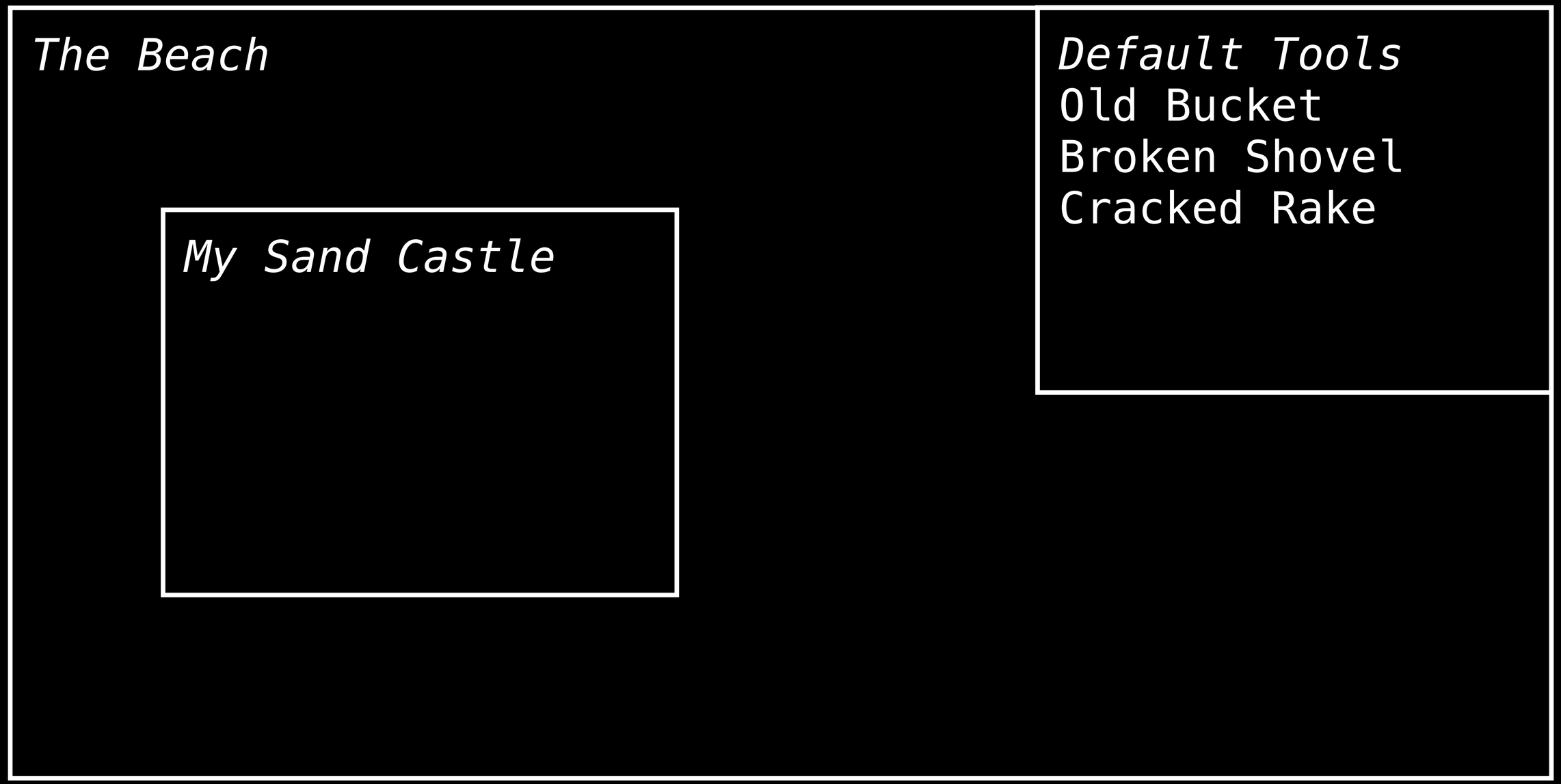
The Beach

Default Tools
Old Bucket
Broken Shovel
Cracked Rake

My Sand Castle
Bucket? Old
Shovel? Broken
Rake? Cracked

I want to build a sand castle! But I don't want to use the default tools.

An Analogy


The Beach
My Sand Castle
Default Tools
Old Bucket
Broken Shovel
Cracked Rake

An Analogy

The Beach

Default Tools
Old Bucket
Broken Shovel
Cracked Rake

My Sand Castle

Idea: Get new tools, store them where you are working, use them instead

An Analogy

The Beach

Default Tools
Old Bucket
Broken Shovel
Cracked Rake

My Sand Castle
New Bucket

Idea: Get new tools, store them where you are working, use them instead

An Analogy

The Beach

Default Tools
Old Bucket
Broken Shovel
Cracked Rake

My Sand Castle
New Bucket
Working Shovel

Idea: Get new tools, store them where you are working, use them instead

An Analogy

The Beach

Default Tools
Old Bucket
Broken Shovel
Cracked Rake

My Sand Castle
New Bucket
Working Shovel
Good Rake

Idea: Get new tools, store them where you are working, use them instead

An Analogy

The Beach

My Sand Castle
New Bucket
Working Shovel
Good Rake

Default Tools
Old Bucket
Broken Shovel
Cracked Rake

Idea: Get new tools, store them where you are working, use them instead

Problem: What if I want to build a new sand castle?

An Analogy

The Beach

Default Tools
Old Bucket
Broken Shovel
Cracked Rake

Idea: A toolshed full of tools, just for my sandcastle business!

An Analogy

The Beach

Default Tools

Sam's Sandcastles

New Bucket
Working Shovel
Good Rake

Idea: A toolshed full of tools, just for my sandcastle business!

An Analogy

The Beach

Default Tools

Sam's Sandcastles

New Bucket
Working Shovel
Good Rake

My Sand Castle

Working on Sam's Sandcastles

Idea: A toolshed full of tools, just for my sandcastle business!

An Analogy

The Beach

Default Tools

Sam's Sandcastles

New Bucket
Working Shovel
Good Rake

My Sand Castle

Working on Sam's Sandcastles

Another Castle

Working on Sam's Sandcastles

Idea: A toolshed full of tools, just for my sandcastle business!

Virtual Environments: I Don't Like Sand

It's coarse and rough and irritating, and it gets everywhere. Not like here. Here everything's soft... and smooth...

The Computer

Default Tools
Python 2
Bad libraries
Old C modules

Idea: A virtual environment, just for CS41: Stanford Python!

Virtual Environments: I Don't Like Sand

It's coarse and rough and irritating, and it gets everywhere. Not like here. Here everything's soft... and smooth...

The Computer

Default Tools

cs41

Python3
iPython
New libraries

Idea: A virtual environment, just for CS41: Stanford Python!

Virtual Environments: I Don't Like Sand

It's coarse and rough and irritating, and it gets everywhere. Not like here. Here everything's soft... and smooth...

The Computer

Default Tools

Assignment 0

Working on CS41

cs41

Python3
iPython
New libraries

Idea: A virtual environment, just for CS41: Stanford Python!

Virtual Environments: I Don't Like Sand

It's coarse and rough and irritating, and it gets everywhere. Not like here. Here everything's soft... and smooth...

The Computer

Default Tools

cs41

Python3
iPython
New libraries

Assignment 0

Working on CS41

Assignment 1

Working on CS41

Idea: A virtual environment, just for CS41: Stanford Python!

A Final Question: How to Get New Tools?

`pip` is the preferred Python package manager

Use `pip`!

When you can, use `pip` instead of:

`conda` - less flexible, less supported by the community

`easy_install` - the old way to install packages

`python setup.py install` - build package from source

Some Advice

Some Advice

Each time you come to the beach, you must point to tools

Some Advice

Each time you come to the beach, you must point to tools

Keeping one set of tools is a good idea

Some Advice

Each time you come to the beach, you must point to tools

Keeping one set of tools is a good idea

Make sure you're using the right tools before building

High Level: Setting Up the Toolshed

High Level: Setting Up the Toolshed

Install Python 3.4

High Level: Setting Up the Toolshed

Install Python 3.4

Install virtualenv + virtualenvwrapper with Python3's **pip**

High Level: Setting Up the Toolshed

Install Python 3.4

Install virtualenv + virtualenvwrapper with Python3's **pip**

Create a virtualenvwrapper for CS41

High Level: Setting Up the Toolshed

Install Python 3.4

Install virtualenv + virtualenvwrapper with Python3's **pip**

Create a virtualenvwrapper for CS41

Install and upgrade packages in that virtual environment.

High Level: Setting Up the Toolshed

Install Python 3.4

Install virtualenv + virtualenvwrapper with Python3's **pip**

Create a virtualenvwrapper for CS41

Install and upgrade packages in that virtual environment.

Detailed instructions to come!

Success looks like…

Success looks like…

Running Python3 from the command line

Success looks like…

Running Python3 from the command line

Activating and deactivating a virtual environment

Success looks like…

Running Python3 from the command line

Activating and deactivating a virtual environment

Installing a package into the virtual environment with pip

Next Time

Transition

Transition

Fewer syntax features, more Python tools and tricks

Transition

Fewer syntax features, more Python tools and tricks

Week 2: Data Structures

Transition

Fewer syntax features, more Python tools and tricks

Week 2: Data Structures

Week 3: Functions

Transition

Fewer syntax features, more Python tools and tricks

Week 2: Data Structures

Week 3: Functions

Week 4: Functional Programming

Buffer: Setup + Lab!

Lab

Time to Experiment!

Make Friends!

CS41 Playlist

Today

Python Installations and Setup

Write some code!