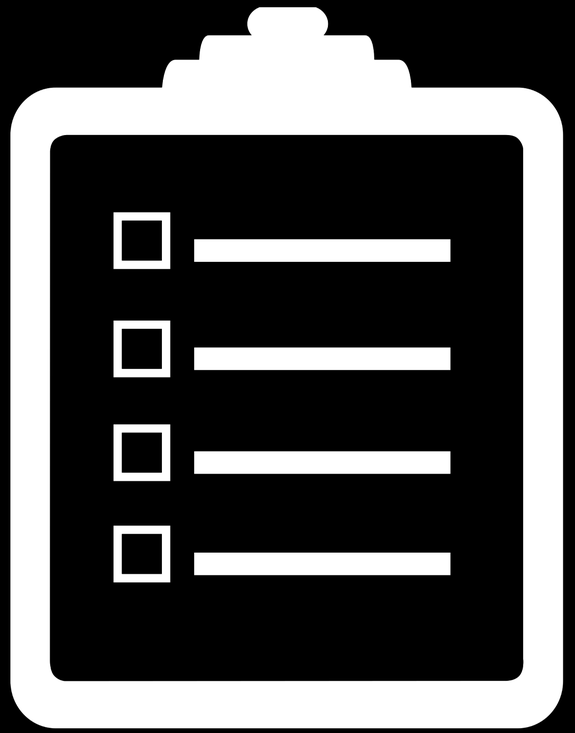
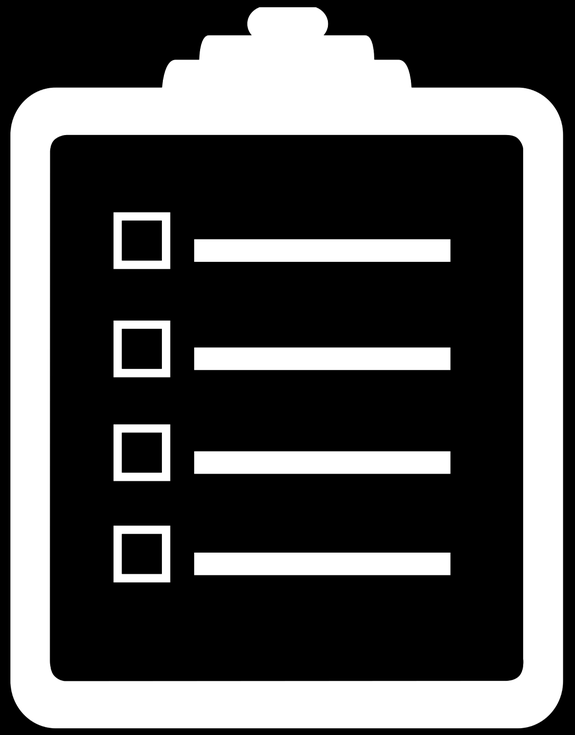
Python Fundamentals

January 10, 2019

Agenda



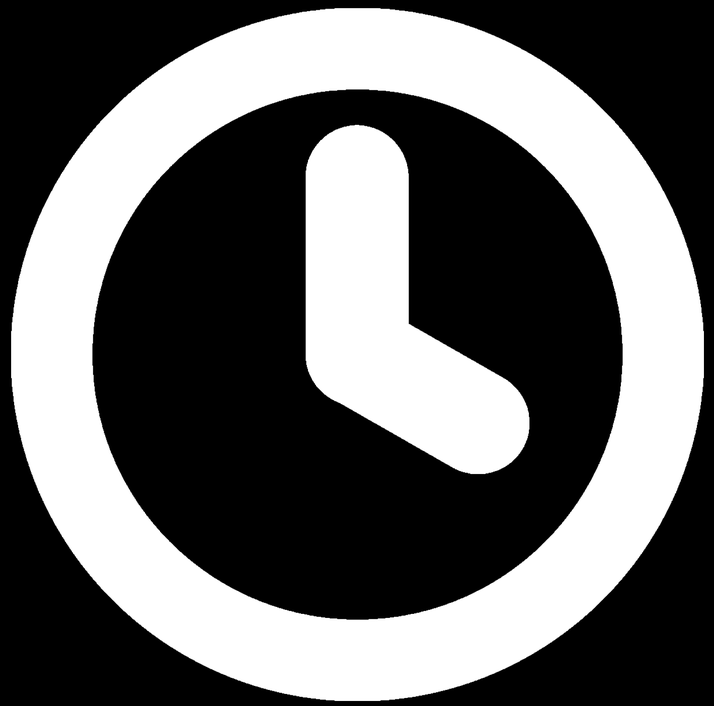
Agenda



- Brief Review
- More Building Blocks
- Types Redux
- String Formatting
- File I/O
- Importing Modules
- Virtual Environments

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.

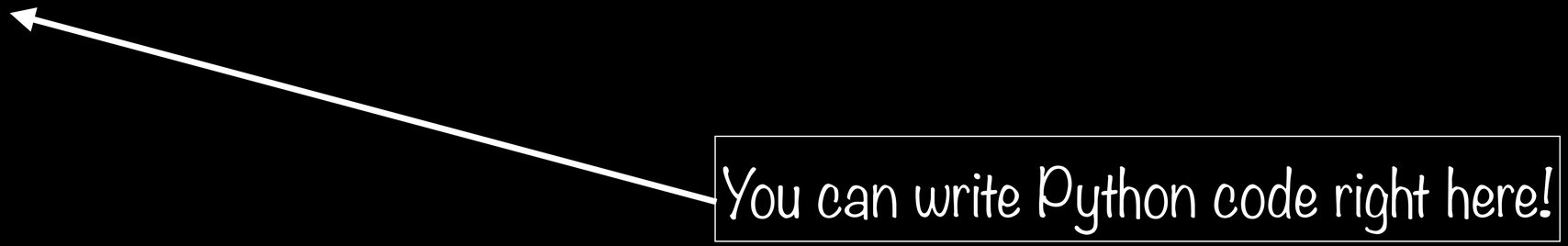
Flat is better than nested.

Sparse is better than dense.

Readability counts.

Interactive Interpreter

```
sredmond$ python3
Python 3.7.2 (default, Dec 27 2018, 07:35:06)
[Clang 10.0.0 (clang-1000.11.45.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```



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
13 / 4          # => 3.25
```

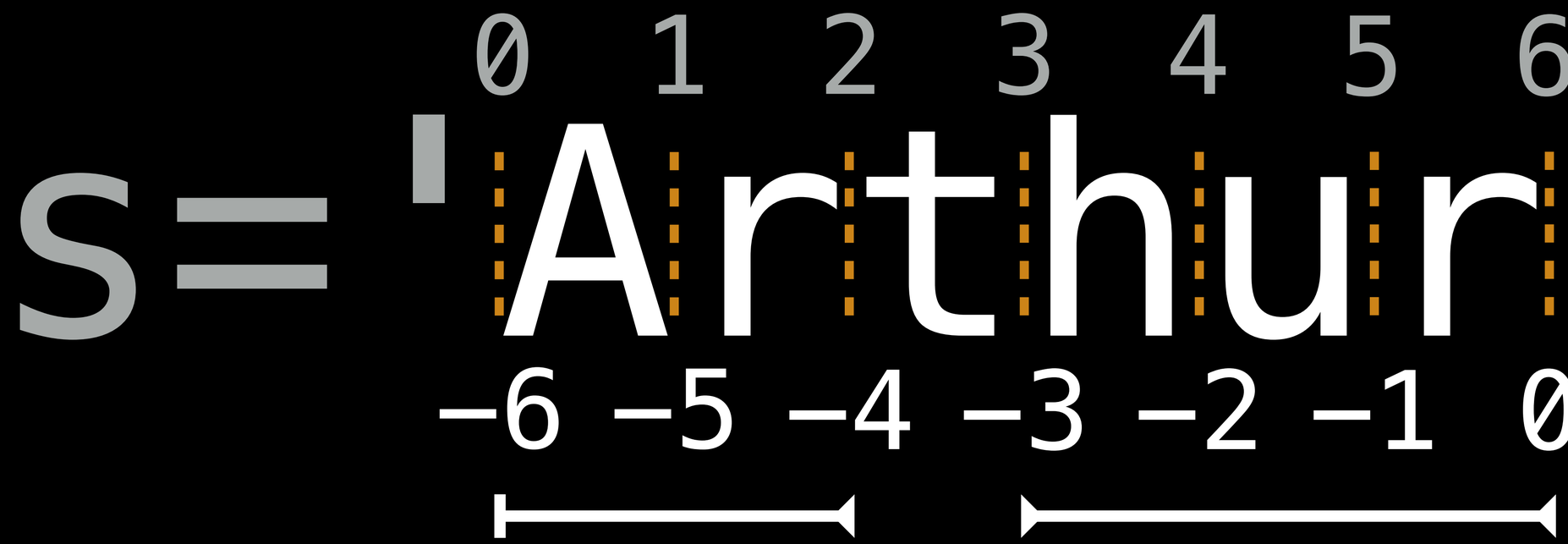
Python has two numeric types `int` and `float`

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`

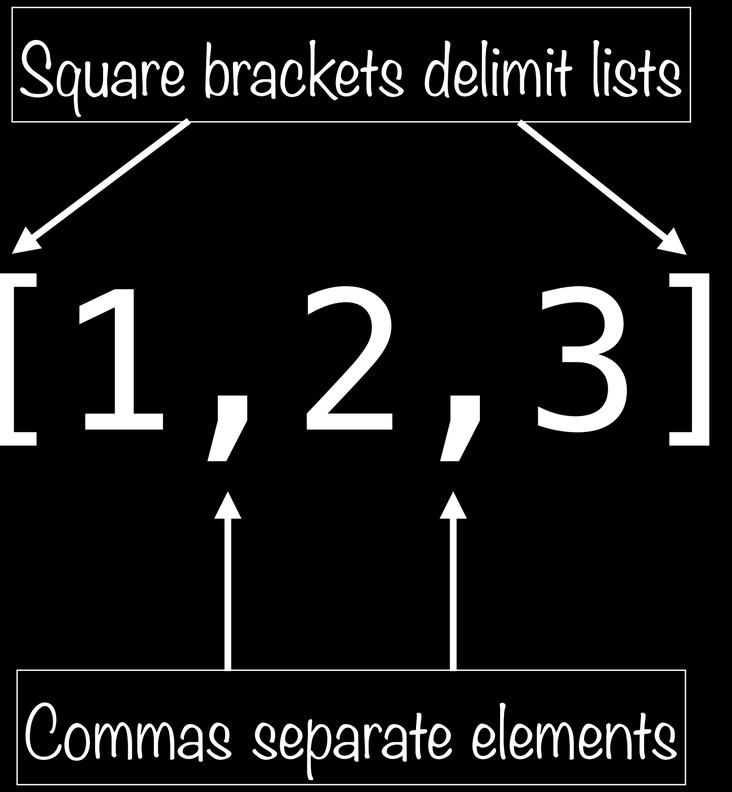
Strings




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

Lists

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



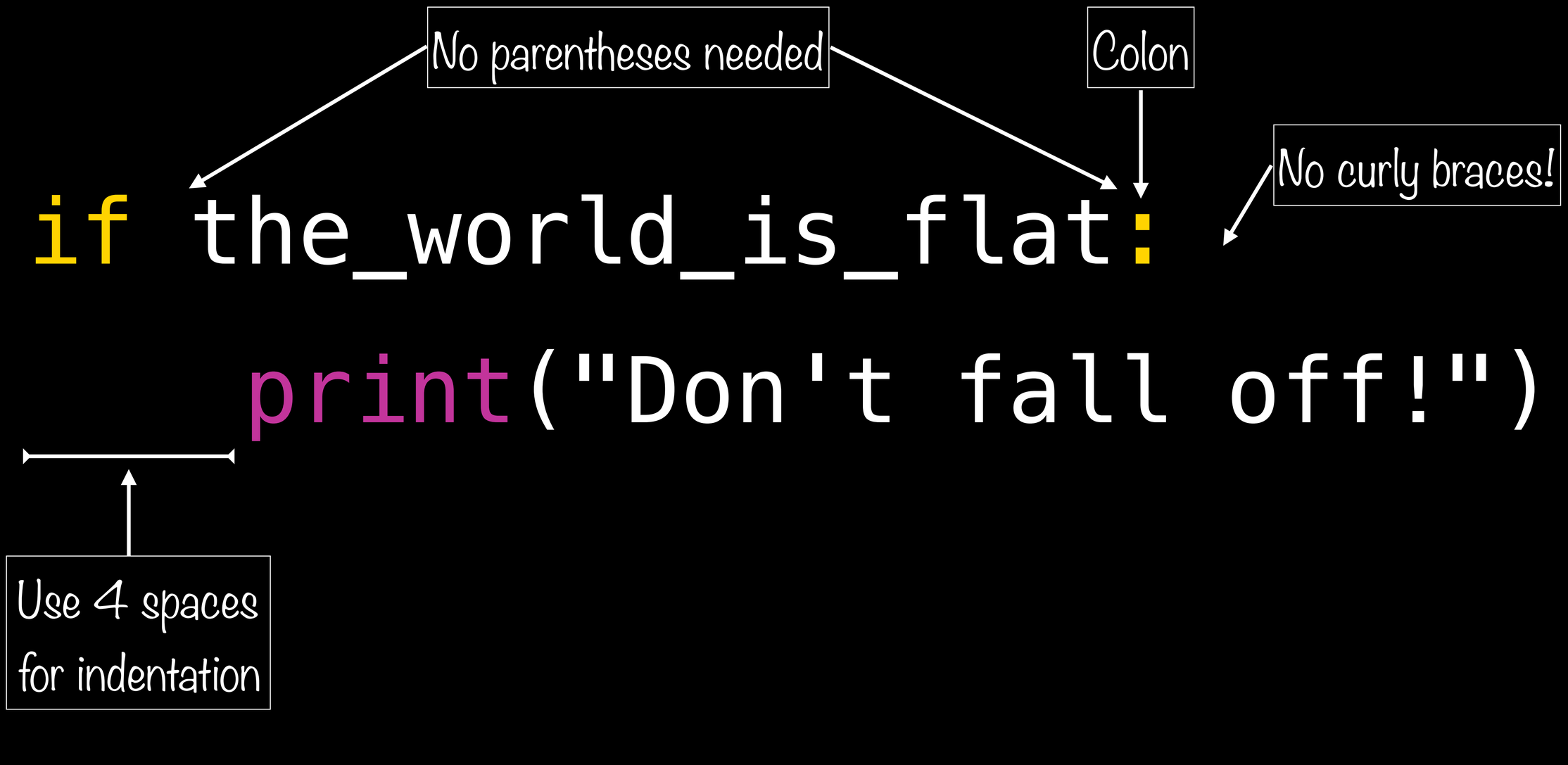
Console I/O

```
# 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
```

`input` prompts the user for input

`print` can be used in many different ways!

If Statements



New Stuff!

Truthy and Falsy

Truthy and Falsy

```
# 'Falsy' values
bool(None)   # => False
bool(False)  # => False
bool(0)      # => False
bool(0.0)    # => False
bool('')     # => False
```

Truthy and Falsy

```
# 'Falsy' values
bool(None)   # => False
bool(False)  # => False
bool(0)      # => False
bool(0.0)    # => False
bool('')     # => False

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

Truthy and Falsy

```
# 'Falsy' values
bool(None)   # => False
bool(False)  # => False
bool(0)      # => False
bool(0.0)    # => False
bool('')     # => False

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

# Everything else is 'truthy'
bool(41)             # => True
bool('abc')          # => True
bool([1, 'a', []])  # => True
```

Truthy and Falsy

```
# 'Falsy' values
bool(None)   # => False
bool(False)  # => False
bool(0)      # => False
bool(0.0)    # => False
bool('')     # => False

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

# Everything else is 'truthy'
bool(41)             # => True
bool('abc')          # => True
bool([1, 'a', []])  # => True

bool([False])  # => True
bool(int)      # => True
```

Checking for Truthiness

with Steven Colbert

Checking for Truthiness

with Steven Colbert

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

Checking for Truthiness

with Steven Colbert

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

Checking for Truthiness

with Steven Colbert

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

Checking for Truthiness

with Steven Colbert

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

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

Checking for Truthiness

with Steven Colbert

```
# 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`

Checking for Truthiness

with Steven Colbert

```
# 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

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

For Loops

Loop explicitly over data

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

For Loops

Loop explicitly over data

Strings, lists, etc.

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

For Loops

Loop explicitly over data

Strings, lists, etc.

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

No loop counter!

Looping over Strings and Lists

Looping over Strings and Lists

```
# Loop over characters in a string.
for ch in "CS41":
    print(ch)
# Prints C, S, 4, and 1
```

Looping over Strings and Lists

```
# Loop over characters in a string.
for ch in "CS41":
    print(ch)
# Prints C, S, 4, and 1
```

Compare

```
String s = "CS41";
for (int i = 0; i < s.length(); ++i) {
    char ch = s.charAt(i);
    System.out.println(ch);
}
```

Looping over Strings and Lists

```
# Loop over characters in a string.
for ch in "CS41":
    print(ch)
# Prints C, S, 4, and 1

# Loop over elements of a list.
for number in [3, 1, 4, 1, 5]:
    print(number ** 2, end='|')
```

Compare

```
String s = "CS41";
for (int i = 0; i < s.length(); ++i) {
    char ch = s.charAt(i);
    System.out.println(ch);
}
```

Looping over Strings and Lists

```
# Loop over characters in a string.
for ch in "CS41":
    print(ch)
# Prints C, S, 4, and 1


# Loop over elements of a list.
for number in [3, 1, 4, 1, 5]:
    print(number ** 2, end='|')
# => 9|1|16|1|25|
```

Compare

```
String s = "CS41";
for (int i = 0; i < s.length(); ++i) {
    char ch = s.charAt(i);
    System.out.println(ch);
}
```

range

Iterate over a sequence of numbers

range

Iterate over a sequence of numbers

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

range

Iterate over a sequence of numbers

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

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

range

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

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

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(2, 10):
    if n == 6:
        break
    print(n, end=', ')
```

`break` and `continue`

```
for n in range(2, 10):
    if n == 6:
        break
    print(n, end=', ')
```

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

`break` and `continue`

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

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

`break` and `continue`

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

for letter in "STELLAR":
    if letter in "LE":
        continue
    print(letter, end='*')
```

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

`break` and `continue`

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

for letter in "STELLAR":
    if letter in "LE":
        continue
    print(letter, end='*')
```

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

`continue` continues with the next iteration of the loop

`break` and `continue`

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

for letter in "STELLAR":
    if letter in "LE":
        continue
    print(letter, end='*')
# => S*T*A*R*
```

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

`continue` continues with the next iteration of the loop

`while` loops

`while` loops

No additional surprises here

`while` loops

No additional surprises here

```
# Print powers of three below 10000
n = 1
while n < 10000:
    print(n)
    n *= 3
```

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

upper_limit = int(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

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

Prime Number Generator

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

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

More to See

More to See

Keyword Arguments

Variadic Argument Lists

Default Argument Values

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
```

Everything is an Object

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

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

isinstance(None, object)       # => True

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

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

Objects have Identity

Objects have Identity

The `id` function returns an object's "identity"

Objects have Identity

The `id` function returns an object's "identity"

```
id(41)   # => 4361704848 (e.g.)
```

Objects have Identity

The `id` function returns an object's "identity"

```
id(41)   # => 4361704848 (e.g.)
```

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

Objects have Identity

The `id` function returns an object's "identity"

```
id(41)   # => 4361704848 (e.g.)
```

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

Objects are tagged with their type at runtime

Objects have Identity

The `id` function returns an object's "identity"

```
id(41)   # => 4361704848 (e.g.)
```

"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

The `id` function returns an object's "identity"

```
id(41)   # => 4361704848 (e.g.)
```

"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 overhead means even small things take up a lot of space!

Objects have Identity

The `id` function returns an object's "identity"

```
id(41)   # => 4361704848 (e.g.)
```

"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 overhead means even small things take up a lot of space!

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

Objects have Identity

The `id` function returns an object's "identity"

```
id(41)  # => 4361704848 (e.g.)
```

"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 overhead means even small things take up a lot of space!

```
(41).__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

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

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
```

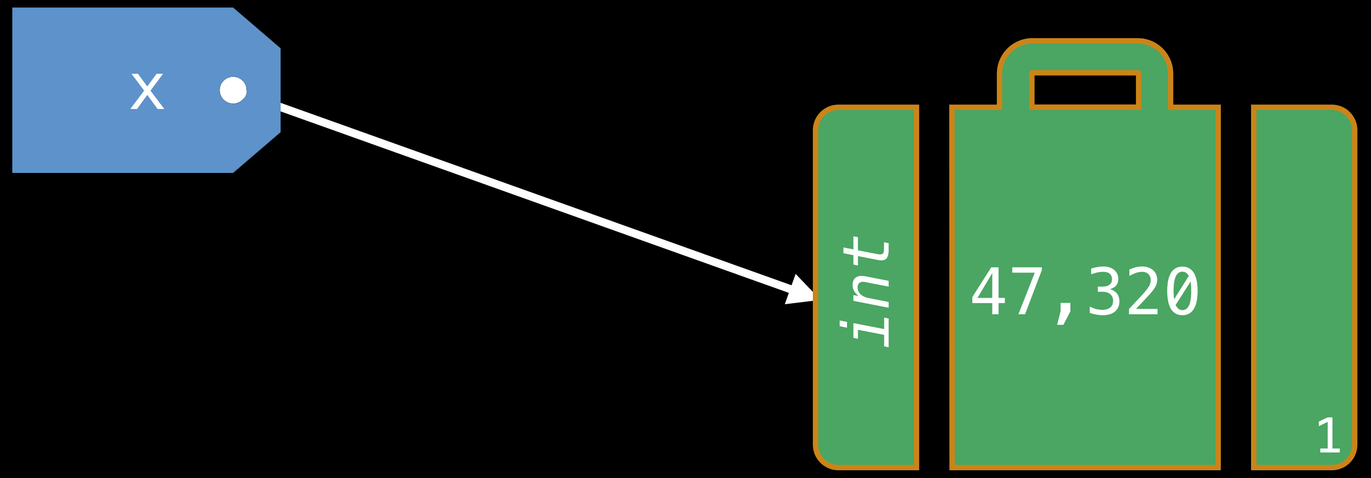
Variables

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!

```
x = 47320
```

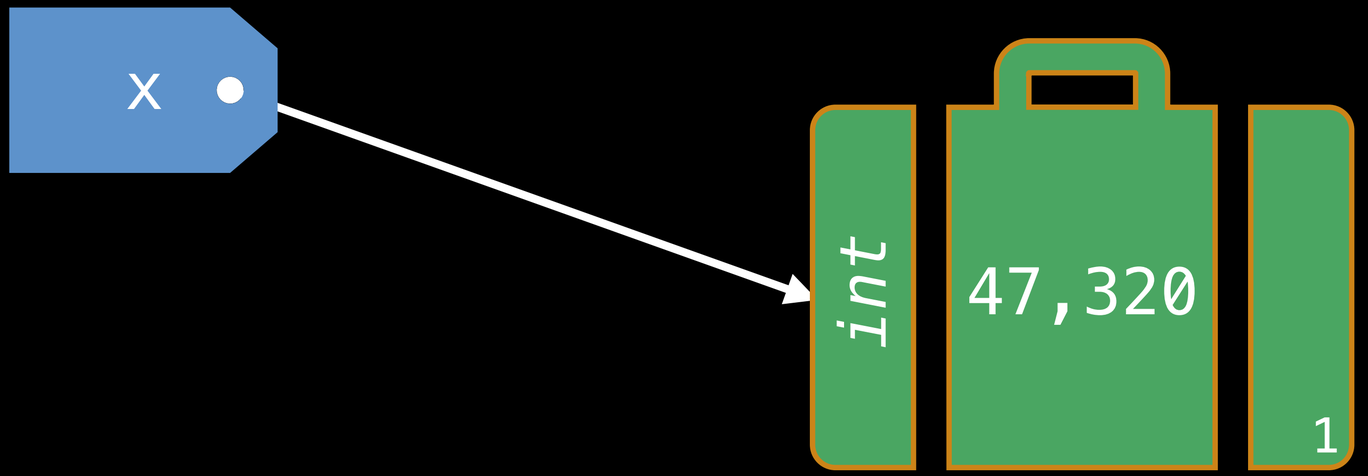
Symbol Table

Remember, "Namespaces are one honking great idea!"

A Python namespace maintains information about labels

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

```
x = 47320
```



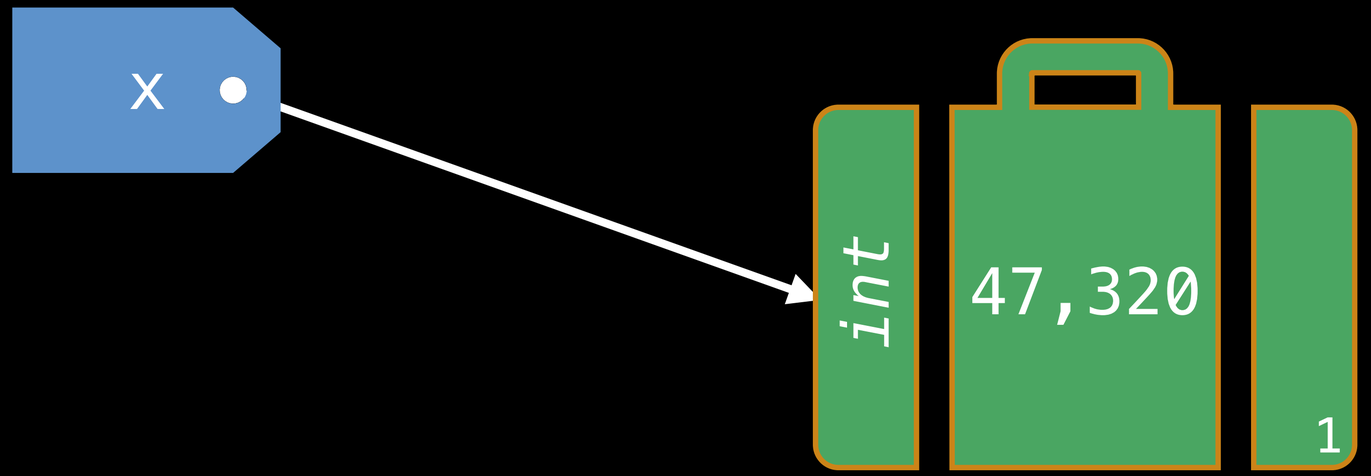
Symbol Table

Remember, "Namespaces are one honking great idea!"

A Python namespace maintains information about labels

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

```
x = 47320

y = "Hello!"
```



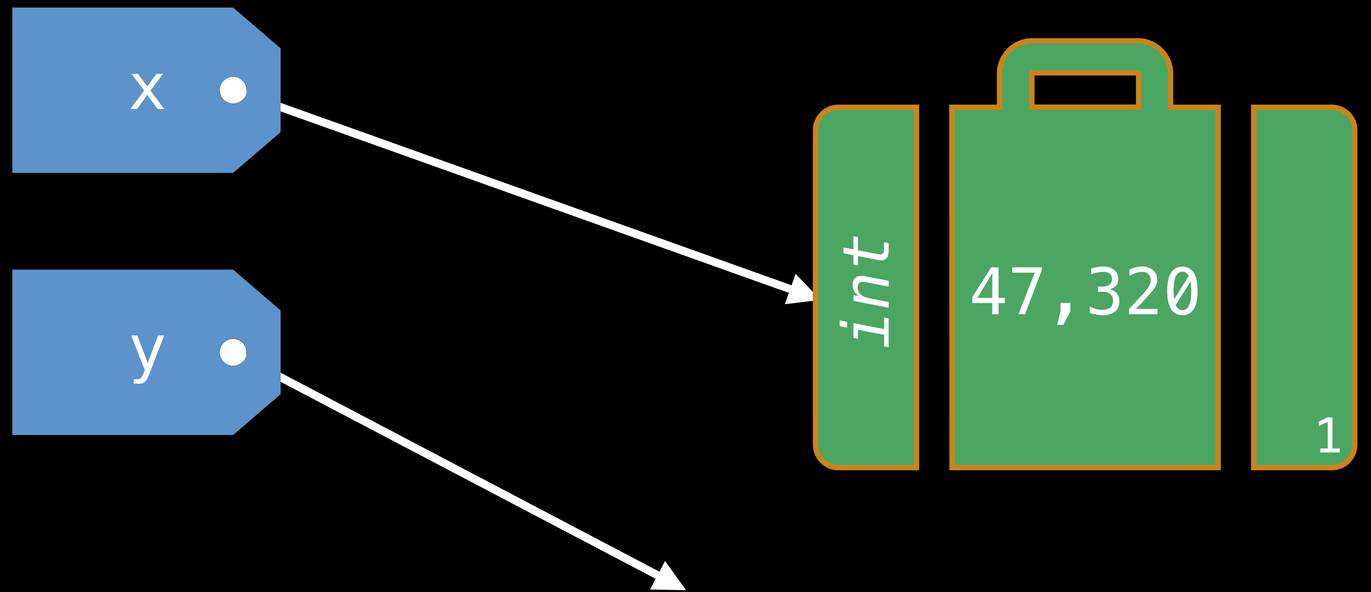
Symbol Table

Remember, "Namespaces are one honking great idea!"

A Python namespace maintains information about labels

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

```
x = 47320

y = "Hello!"
```

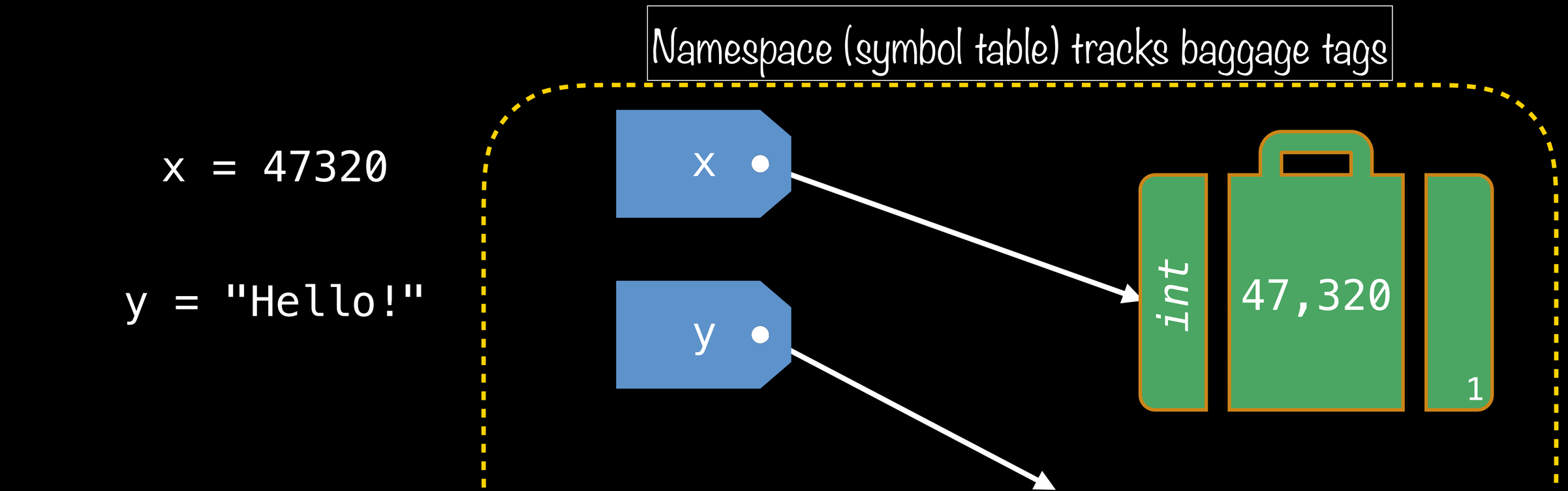


Symbol Table

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

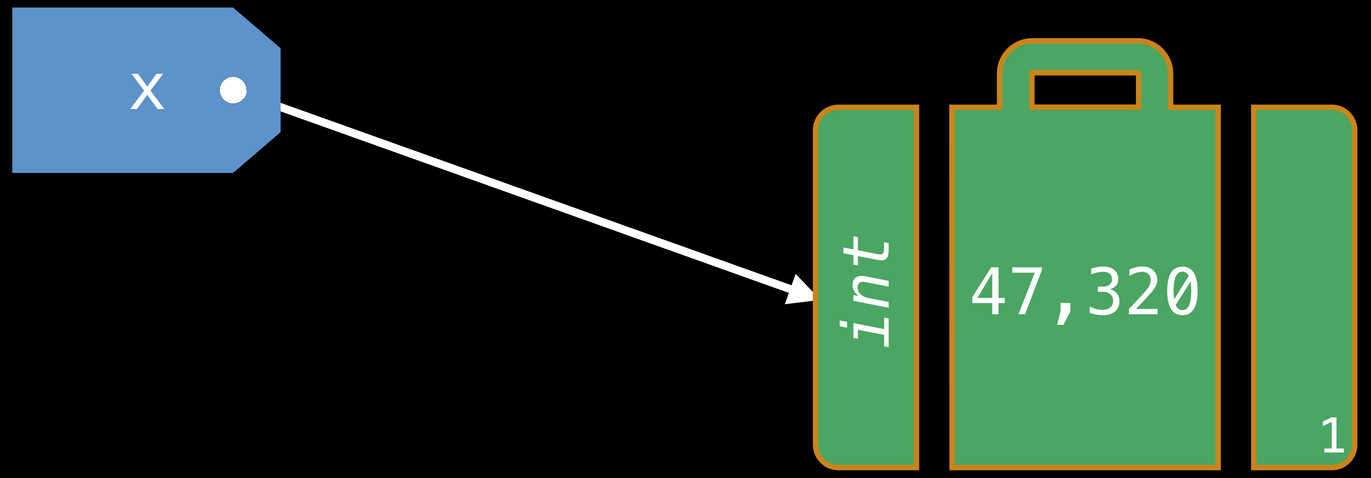
Assigning from a variable does **not** copy an object.

```
x = 47320
```

Variable Assignment

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

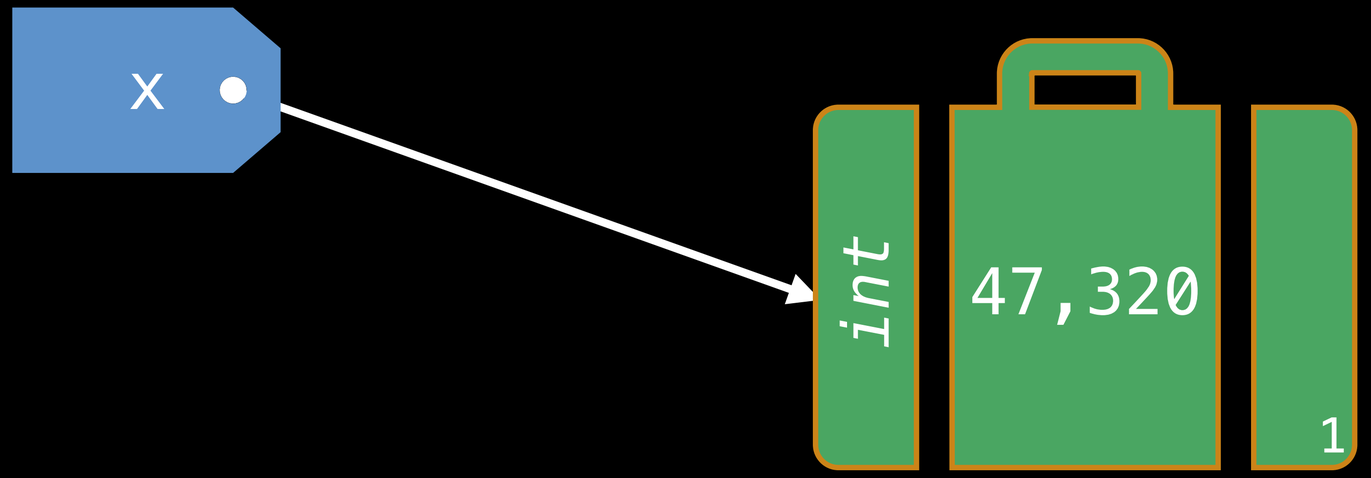
```
x = 47320
```



Variable Assignment

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

```
x = 47320

y = x
```

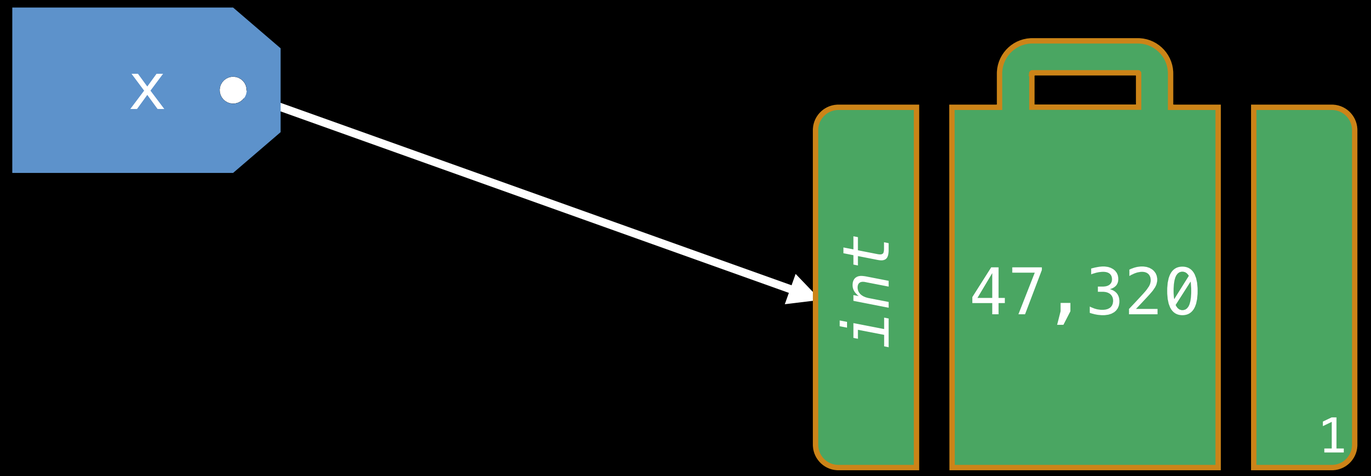


Variable Assignment

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

Instead, it adds another reference to the same object.

```
x = 47320

y = x
```

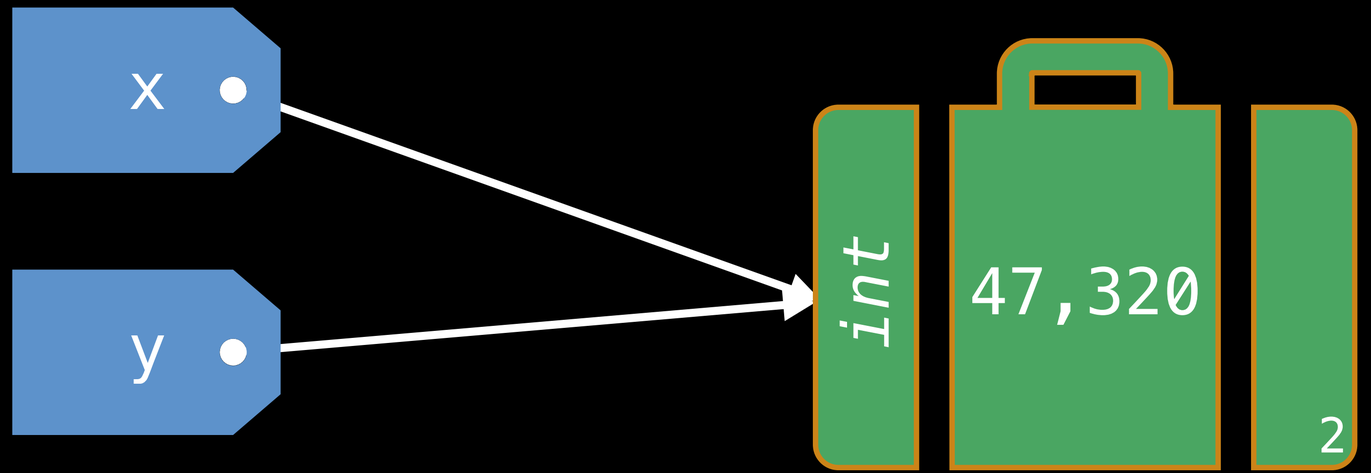


Variable Assignment

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

Instead, it adds another reference to the same object.

```
x = 47320

y = x
```



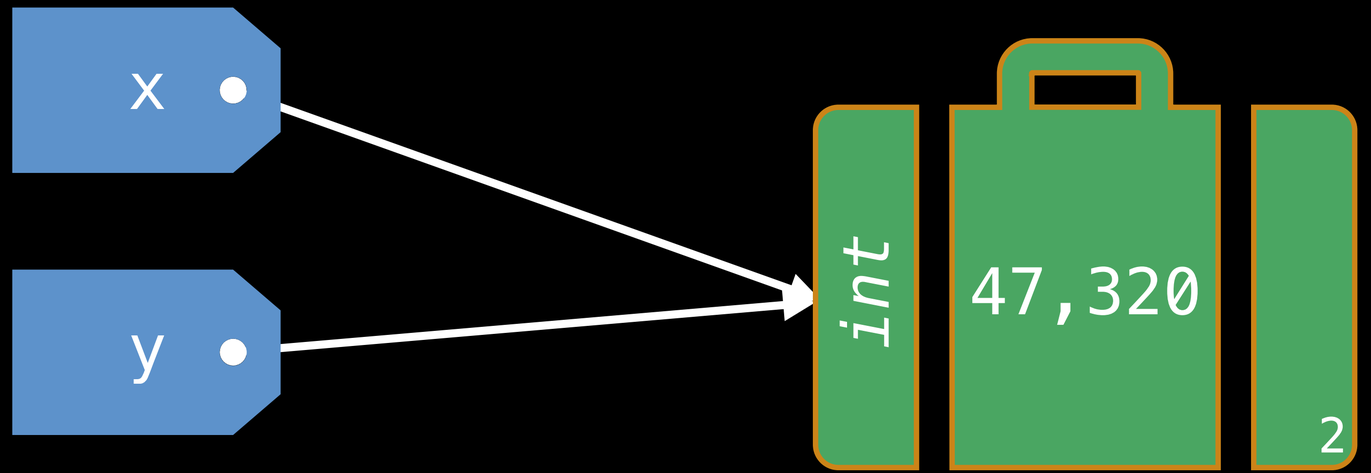
Variable Assignment

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(4, 1, 3)  # => 15
```

Duck Typing

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

compute(4, 1, 3)  # => 15
compute([1], [2, 3], 2) # => [1, 2, 3, 1, 2, 3]
```

Duck Typing

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

compute(4, 1, 3)  # => 15
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(4, 1, 3)  # => 15
compute([1], [2, 3], 2) # => [1, 2, 3, 1, 2, 3]
compute('l', 'olo', 4)  # => 'lololololololololo'
```

To the `compute` function, 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
```

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
```

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

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

Identity Crisis

Identity Crisis

```
x = "hello world"
```

Identity Crisis

```
x = "hello world"
y = "hello "
y += "world"
```

Identity Crisis

```
x = "hello world"
y = "hello "
y += "world"

x == y  # => True
x is y  # => False
```

Identity Crisis

```
x = "hello world"
y = "hello "
y += "world"

x == y  # => True
x is y  # => False

id(x)  # => 4455464720
id(y)  # => 4455960848
```

Identity Crisis

```
x = "hello world"
y = "hello "
y += "world"

x == y  # => True
x is y  # => False

id(x)  # => 4455464720
id(y)  # => 4455960848

4 > 3 is not True  # What's the output?
```

a is not b

is syntactic sugar for

not (a is b)

Identity Crisis

```
x = "hello world"
y = "hello "
y += "world"

x == y  # => True
x is y  # => False

id(x)  # => 4455464720
id(y)  # => 4455960848

4 > 3 is not True  # What's the output?
# => True
```

a is not b

is syntactic sugar for

not (a is b)

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.
```

Just 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! "
greeting.upper()  # => "HELLO WORLD! "
```

Useful String Methods

```
greeting = "Hello world! "

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

greeting.strip()  # => "Hello world!"
greeting.strip('dH !')  # => "ello worl"
```

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 (of strings)
', '.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('left')  # => 'left      '
'{:*^12}'.format('CS41')  # => '****CS41****'
```

Fancier Formatting

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

# Padding is just another specifier.
'{:10}'.format('left')  # => 'left      '
'{:*^12}'.format('CS41')  # => '****CS41****'

# 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('left')  # => 'left      '
'{:*^12}'.format('CS41')  # => '****CS41****'

# 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: Hard to read, duplicated values

Aside: Other Formatting Approaches

Problem: Hard to read, duplicated values

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

# 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: Hard to read, duplicated values

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

Problem: Slow, 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: Hard to read, duplicated values

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

Problem: Slow, explicit string conversion

```
# Newfangled formatted string literals
f"I am {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: Hard to read, duplicated values

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

Problem: Slow, explicit string conversion

```
# Newfangled formatted string literals
f"I am {age} years old."
```

Problem: Uncommon, only on 3.6+

Aside: Other Formatting Approaches

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

Problem: Hard to read, duplicated values

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

Problem: Slow, explicit string conversion

```
# Newfangled formatted string literals
f"I am {age} years old."
```

Problem: Uncommon, only on 3.6+

Using `.format()` is probably the right choice

Time-Out for Announcements

Announcements

Announcements

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

Announcements

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

Auditors Email us so we add you to our internal lists.

Announcements

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

Auditors Email us so we add you to our internal lists.

Axess Enrollment codes!

Announcements

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

Auditors Email us so we add you to our internal lists.

Axess Enrollment codes!

Attendance Keep using iamhere.stanfordpython.com

Announcements

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

Auditors Email us so we add you to our internal lists.

Axess Enrollment codes!

Attendance Keep using iamhere.stanfordpython.com

Lecture Material Slides always, videos with best effort.

Announcements

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

Auditors Email us so we add you to our internal lists.

Axess Enrollment codes!

Attendance Keep using iamhere.stanfordpython.com

Lecture Material Slides always, videos with best effort.

Assignment 0 Warm up, check installation + submission.

Back to Python!

File I/O

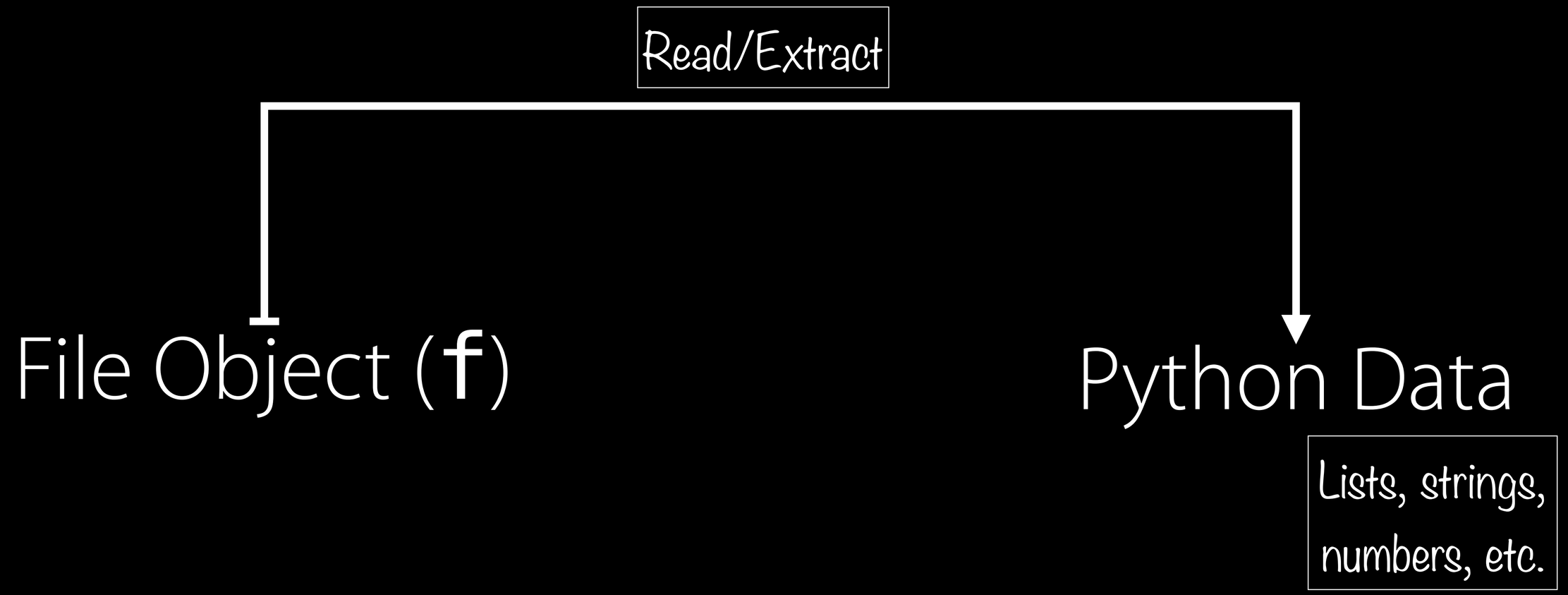
File I/O

File Object (**f**)

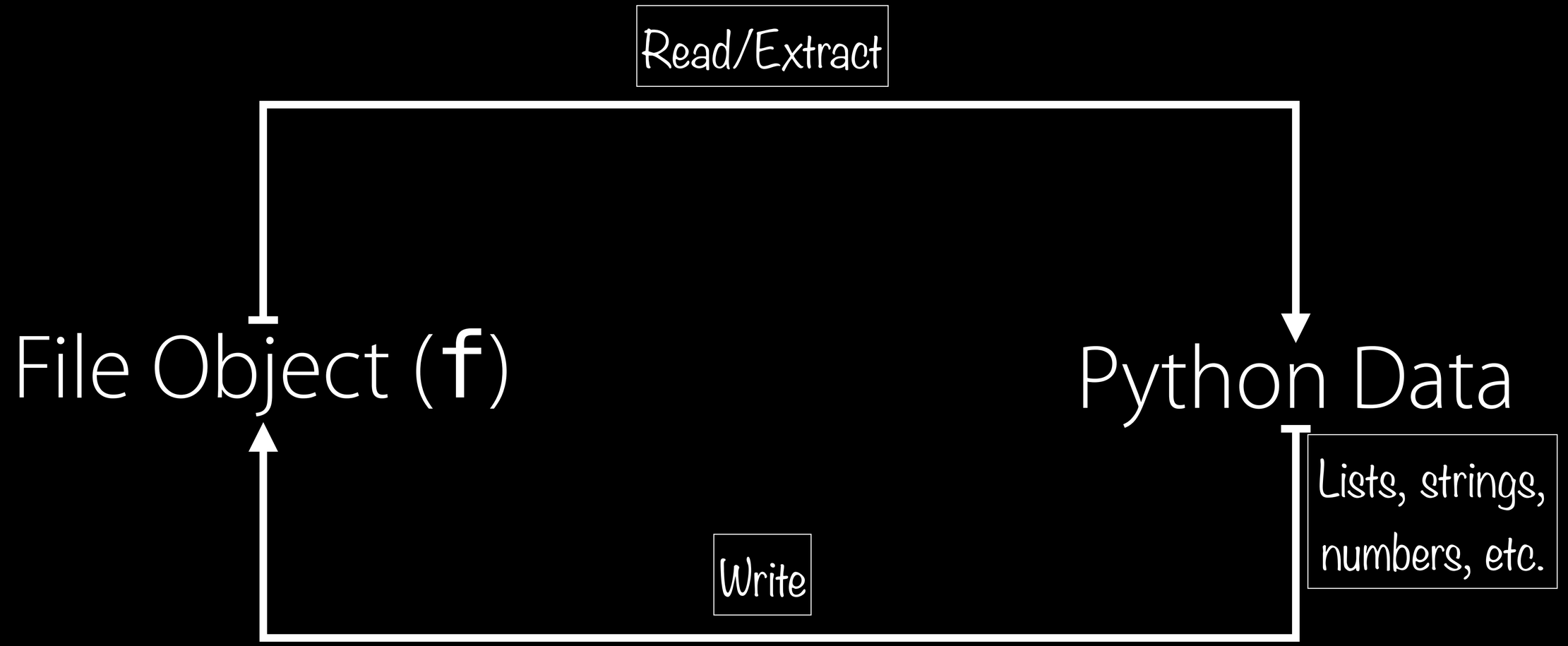
Python Data

Lists, strings, numbers, etc.

File I/O


Read/Extract
File Object (f)
Python Data
Lists, strings,
numbers, etc.

File I/O


Read/Extract
File Object (f)
Python Data
Lists, strings, numbers, etc.
Write

File I/O

File Object (**f**)

Python Data

```
open(filename, mode)
```

File I/O

File Object (**f**)

Python Data

File I/O

```
open(filename, mode)
```

File Object (**f**)

Python Data

File I/O

```
open(filename, mode)
```

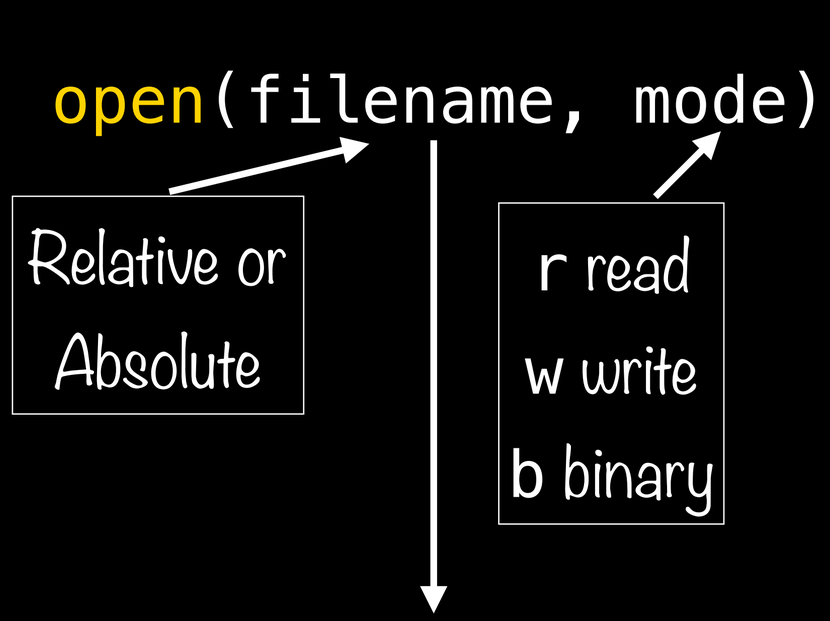
Relative or Absolute

r read
w write
b binary

File Object (**f**)

Python Data

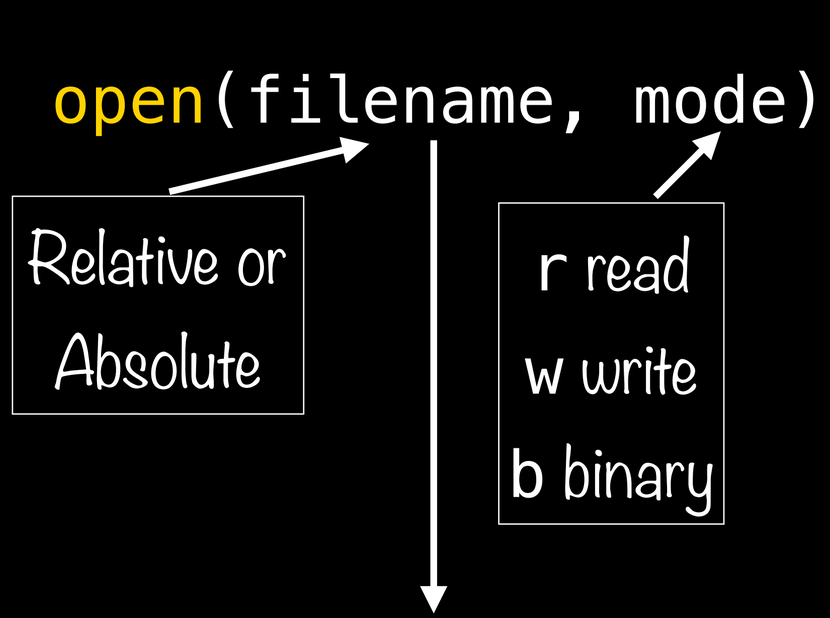
File I/O



File Object (f)

Python Data

File I/O

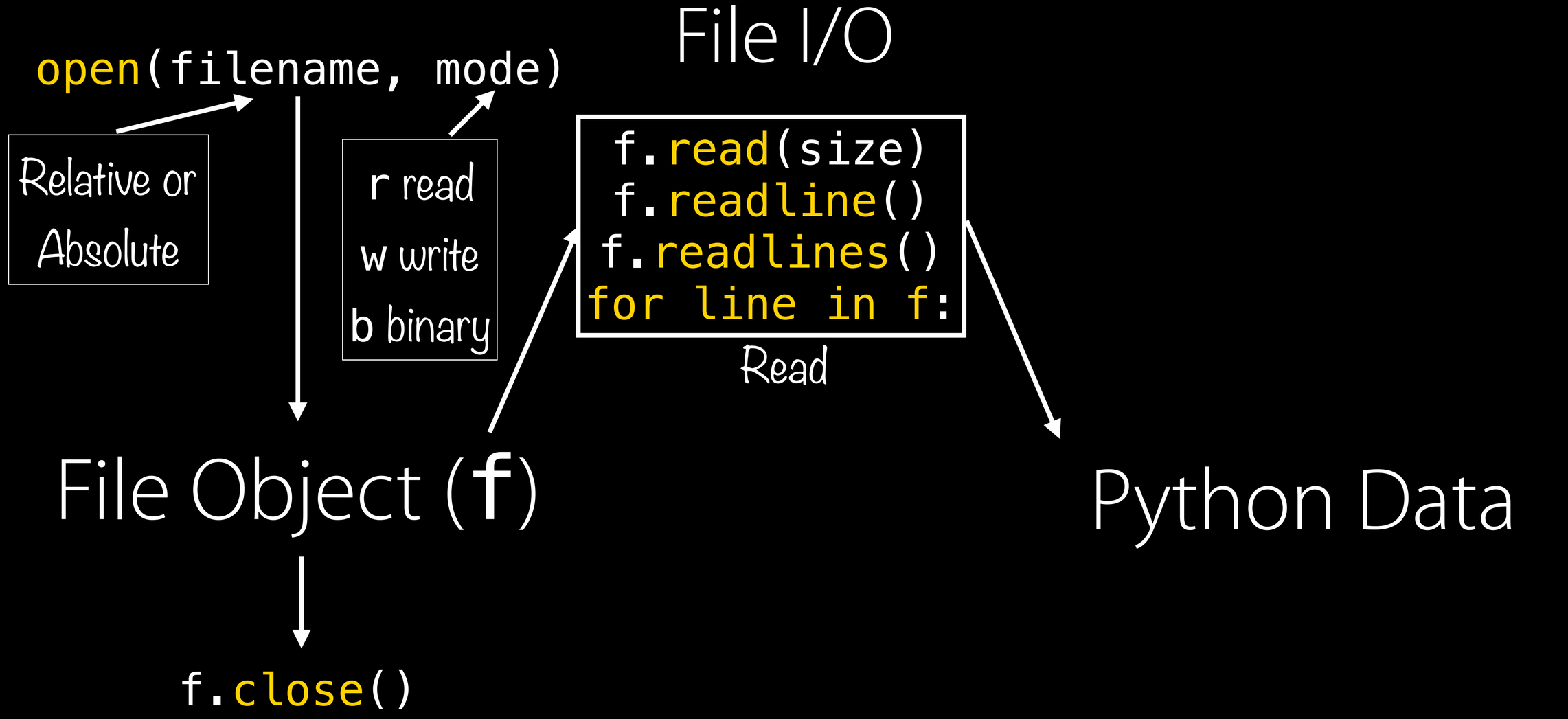


File Object (f)

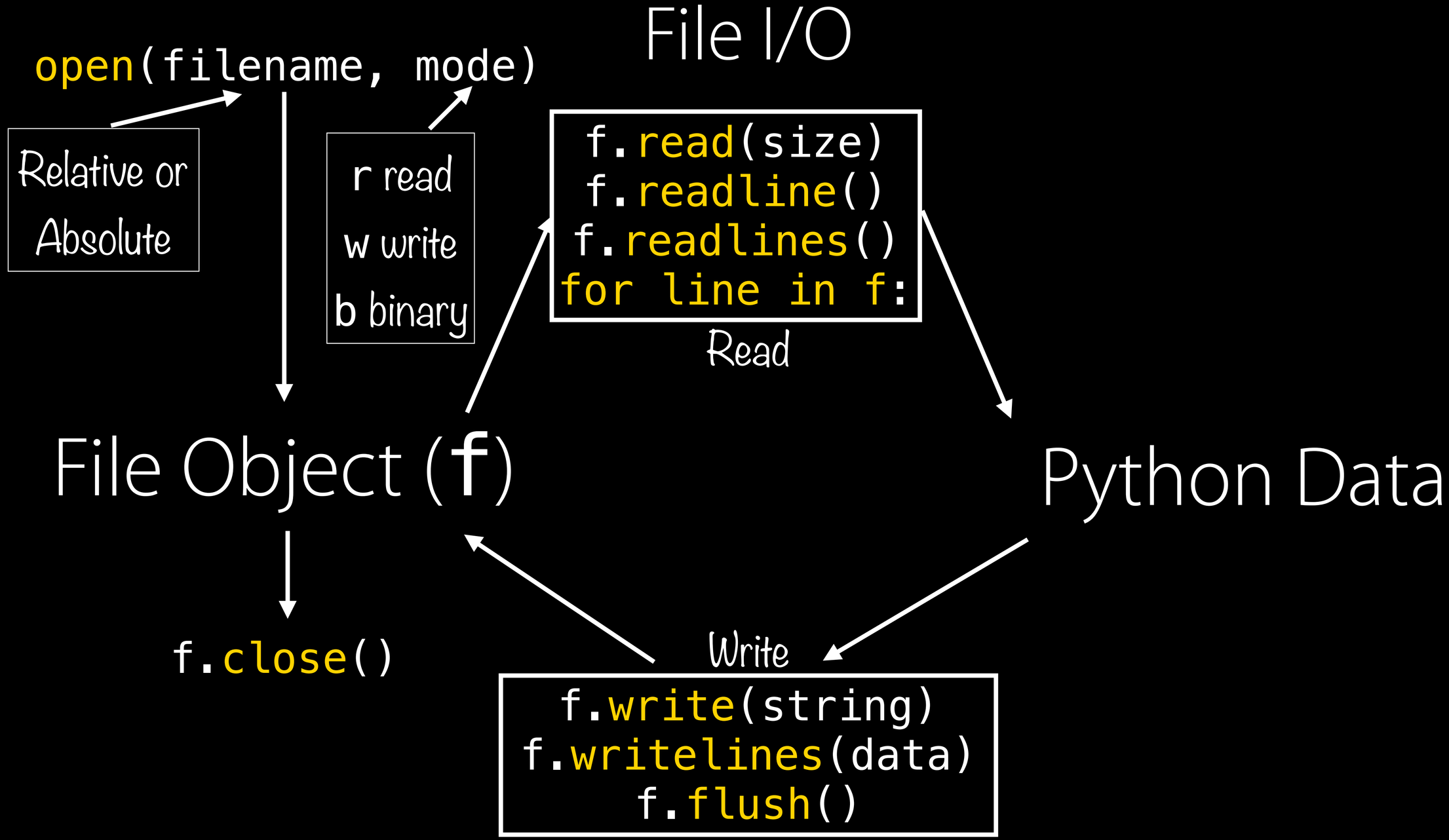
```
f.close()
```

Python Data

File I/O


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)
f.close()
Python Data

File I/O


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()

File Reading

File Reading

```
f = open("knights.txt")
```

File Reading

```
f = open("knights.txt")
```

Suppose `knights.txt` tracks a knight's jousting wins and losses

```
Lancelot 6 0
Galahad 7 12
Geraint 3 1
Mordred 0 0
```

knights.txt

File Reading

```
f = open("knights.txt")
for line in f:
```

Suppose `knights.txt` tracks a knight's jousting wins and losses

```
Lancelot 6 0
Galahad 7 12
Geraint 3 1
Mordred 0 0
```

knights.txt

File Reading

```
f = open("knights.txt")
for line in f:
    data = line.split(' ')
```

Suppose `knights.txt` tracks a knight's jousting wins and losses

```
Lancelot 6 0
Galahad 7 12
Geraint 3 1
Mordred 0 0
```

knights.txt

File Reading

```
f = open("knights.txt")
for line in f:
    data = line.split(' ')

    name = data[0]
    wins = int(data[1])
    losses = int(data[2])
```

Suppose `knights.txt` tracks a knight's jousting wins and losses

```
Lancelot 6 0
Galahad 7 12
Geraint 3 1
Mordred 0 0
```

knights.txt

File Reading

```
f = open("knights.txt")
for line in f:
    data = line.split(' ')

    name = data[0]
    wins = int(data[1])
    losses = int(data[2])
```

Suppose `knights.txt` tracks a knight's jousting wins and losses

```
Lancelot 6 0
Galahad 7 12
Geraint 3 1
Mordred 0 0
```

knights.txt

We'll see better ways to unpack data later

File Reading

```
f = open("knights.txt")
for line in f:
    data = line.split(' ')

    name = data[0]
    wins = int(data[1])
    losses = int(data[2])

    win_percent = 100 * wins / (wins + losses)
```

Suppose `knights.txt` tracks a knight's jousting wins and losses

```
Lancelot 6 0
Galahad 7 12
Geraint 3 1
Mordred 0 0
```

knights.txt

We'll see better ways to unpack data later

File Reading

```
f = open("knights.txt")
for line in f:
    data = line.split(' ')

    name = data[0]
    wins = int(data[1])
    losses = int(data[2])

    win_percent = 100 * wins / (wins + losses)
    print(f"{name}: Wins {win_percent:.2f}%")
```

Suppose `knights.txt` tracks a knight's jousting wins and losses

```
Lancelot 6 0
Galahad 7 12
Geraint 3 1
Mordred 0 0
```

knights.txt

We'll see better ways to unpack data later

File Reading

```
f = open("knights.txt")
for line in f:
    data = line.split(' ')

    name = data[0]
    wins = int(data[1])
    losses = int(data[2])

    win_percent = 100 * wins / (wins + losses)
    print(f"{name}: Wins {win_percent:.2f}%")
f.close()
```

Suppose `knights.txt` tracks a knight's jousting wins and losses

```
Lancelot 6 0
Galahad 7 12
Geraint 3 1
Mordred 0 0
```

knights.txt

We'll see better ways to unpack data later

File Reading

```
f = open("knights.txt")
for line in f:
    data = line.split(' ')

    name = data[0]
    wins = int(data[1])
    losses = int(data[2])

    win_percent = 100 * wins / (wins + losses)
    print(f"{name}: Wins {win_percent:.2f}%")
f.close()
```

Suppose `knights.txt` tracks a knight's jousting wins and losses

```
Lancelot 6 0
Galahad 7 12
Geraint 3 1
Mordred 0 0
```

knights.txt

We'll see better ways to unpack data later

Something goes wrong here...

What if…?

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

What if…?

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

The file is never closed! That's bad!

Be Responsible: Use Context Managers

Be Responsible: Use Context Managers

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

Be Responsible: Use Context Managers

```
with open('knights.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: Use Context Managers

```
with open('knights.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: Use Context Managers

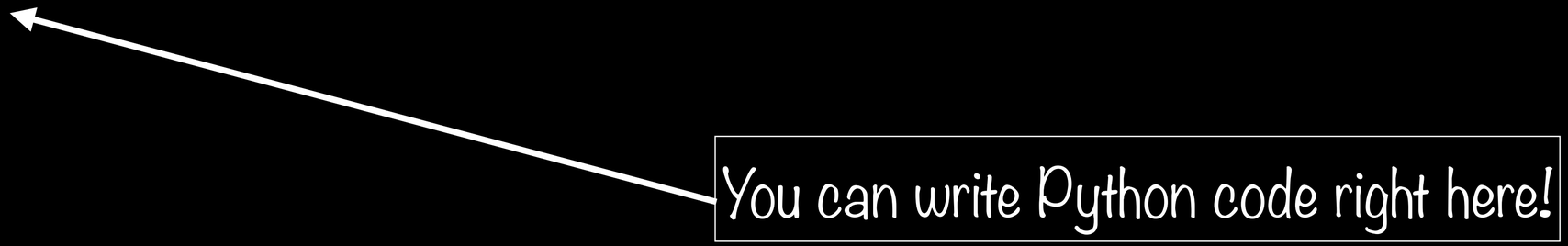
```
with open('knights.txt', 'r') as f:
    content = f.read()
    print(1/0)


f.closed  # => True

# `content` is still in scope
'content' in locals()
```

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 3.7.2 (default, Dec 27 2018, 07:35:06)
[Clang 10.0.0 (clang-1000.11.45.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```



Interactive Interpreter: Great, but…

Interactive Interpreter: Great, but…

Problem Temporary

Interactive Interpreter: Great, but…

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

Supply the filename of the Python script to run after the `python3` command

Running Python Scripts

```
sredmond$ python3 my_script.py
```

Supply the filename of the Python script to run after the `python3` command

Running Python Scripts

```
sredmond$ python3 my_script.py
<output from the script>
```

Supply the filename of the Python script to run after the `python3` command

Running Python Scripts

```
sredmond$ python3 my_script.py
<output from the script>

sredmond$ python3 hello.py
```

Supply the filename of the Python script to run after the `python3` command

Running Python Scripts

```
sredmond$ python3 my_script.py
<output from the script>
```

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 my_script.py
<output from the script>

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

sredmond$
```

Supply the filename of the Python script to run after the `python3` command

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("Joy")
Hey Joy, I'm Python!
>>>
```

Executable Scripts

Executable Scripts

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

We can make the Python script executable with chmod,
as long as the shebang line specified a Python interpreter

Executable Scripts

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

We can make the Python script executable with chmod,
as long as the shebang line specified a Python interpreter

Executable Scripts

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

We can make the Python script executable with chmod,
as long as the shebang line specified a Python interpreter

Executable Scripts

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

We can make the Python script executable with chmod, as long as the shebang line specified a Python interpreter

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_file 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_file import my_function, my_variable
```

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

Virtual Environments Primer

Local, isolated Python environment for interpreter, third-party libraries, and scripts

Problem

Problem

Imagine one application requires LibFoo v1 and another uses LibFoo v2.

We'll need local, isolated execution environments.

Problem

Imagine one application requires LibFoo v1 and another uses LibFoo v2.

We'll need local, isolated execution environments.

We use Python 3 but many computers default to Python 2.7.

Different projects / courses may require different library versions.

Problem

Imagine one application requires LibFoo v1 and another uses LibFoo v2.

We'll need local, isolated execution environments.

We use Python 3 but many computers default to Python 2.7.

Different projects / courses may require different library versions.

Solution: Create an isolated sandbox for all of this course's Python stuff.

We will use an isolated Python environment for CS41!

An Analogy

An Analogy

The Beach

An Analogy

An Analogy

The Beach

Default Toolshed

Old Bucket
Broken Shovel
Cracked Rake

An Analogy

The Beach

Default Toolshed

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 Toolshed

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

Bucket? Old

Default Toolshed

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 Toolshed

```
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 Toolshed

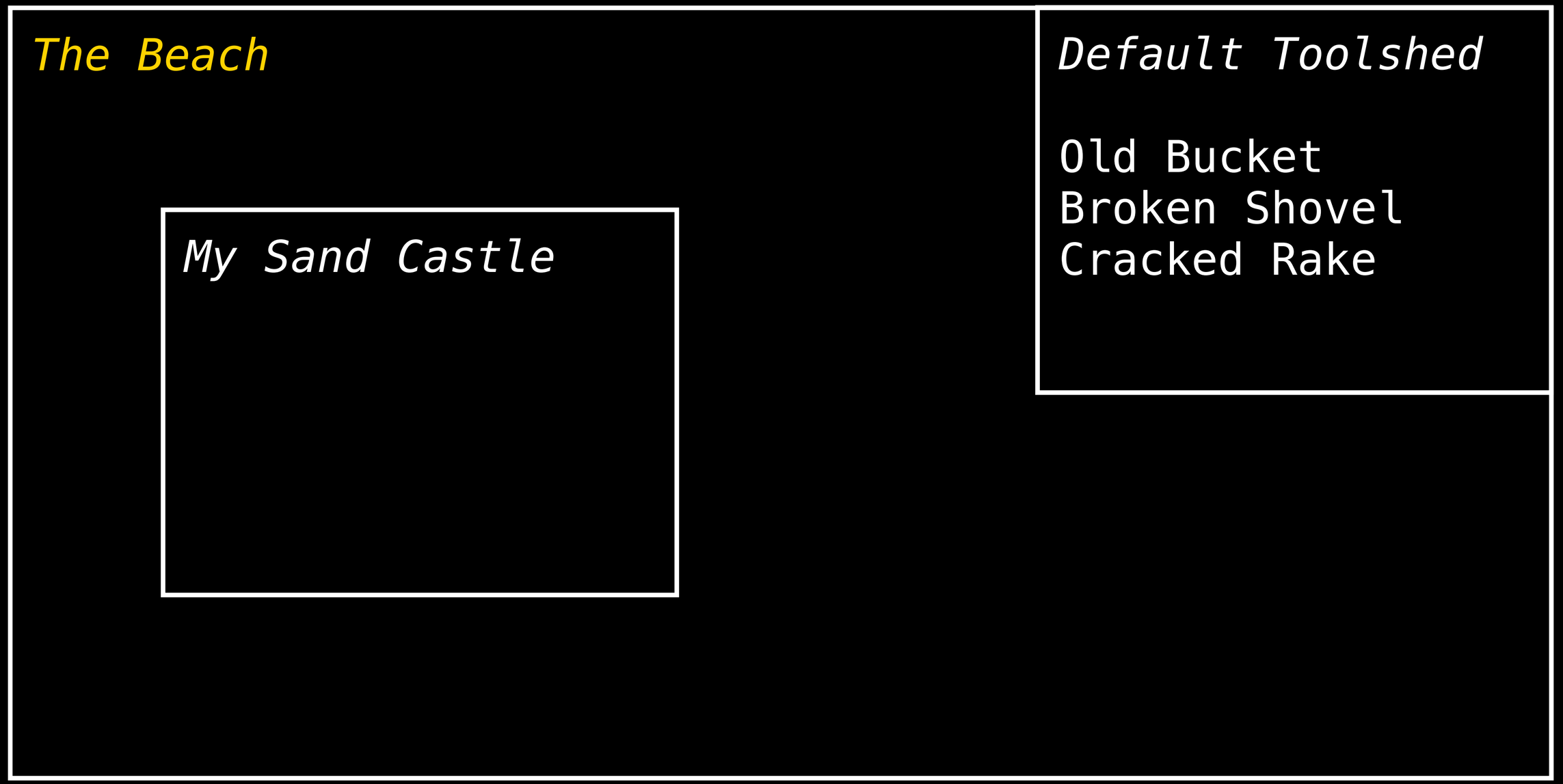
Old Bucket
Broken Shovel
Cracked Rake

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 Toolshed
Old Bucket
Broken Shovel
Cracked Rake

An Analogy

The Beach

My Sand Castle

Default Toolshed

Old Bucket
Broken Shovel
Cracked Rake

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

An Analogy

The Beach

My Sand Castle

New Bucket

Default Toolshed

Old Bucket
Broken Shovel
Cracked 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

Default Toolshed

Old Bucket
Broken Shovel
Cracked 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 Toolshed

Old Bucket
Broken Shovel
Cracked 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 Toolshed

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 Toolshed

Old Bucket
Broken Shovel
Cracked Rake

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

An Analogy

The Beach

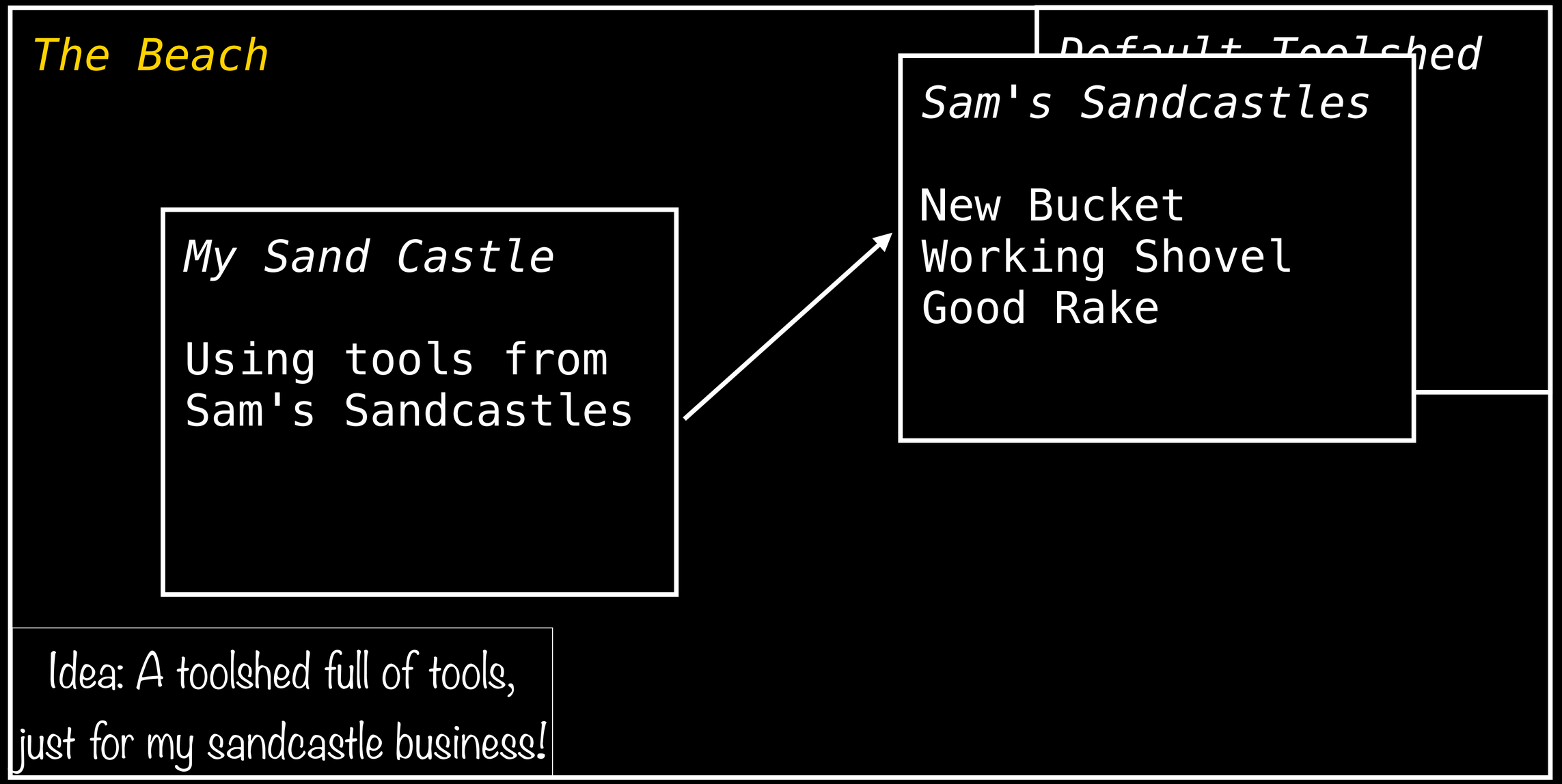
Default Toolshed

Sam's Sandcastles

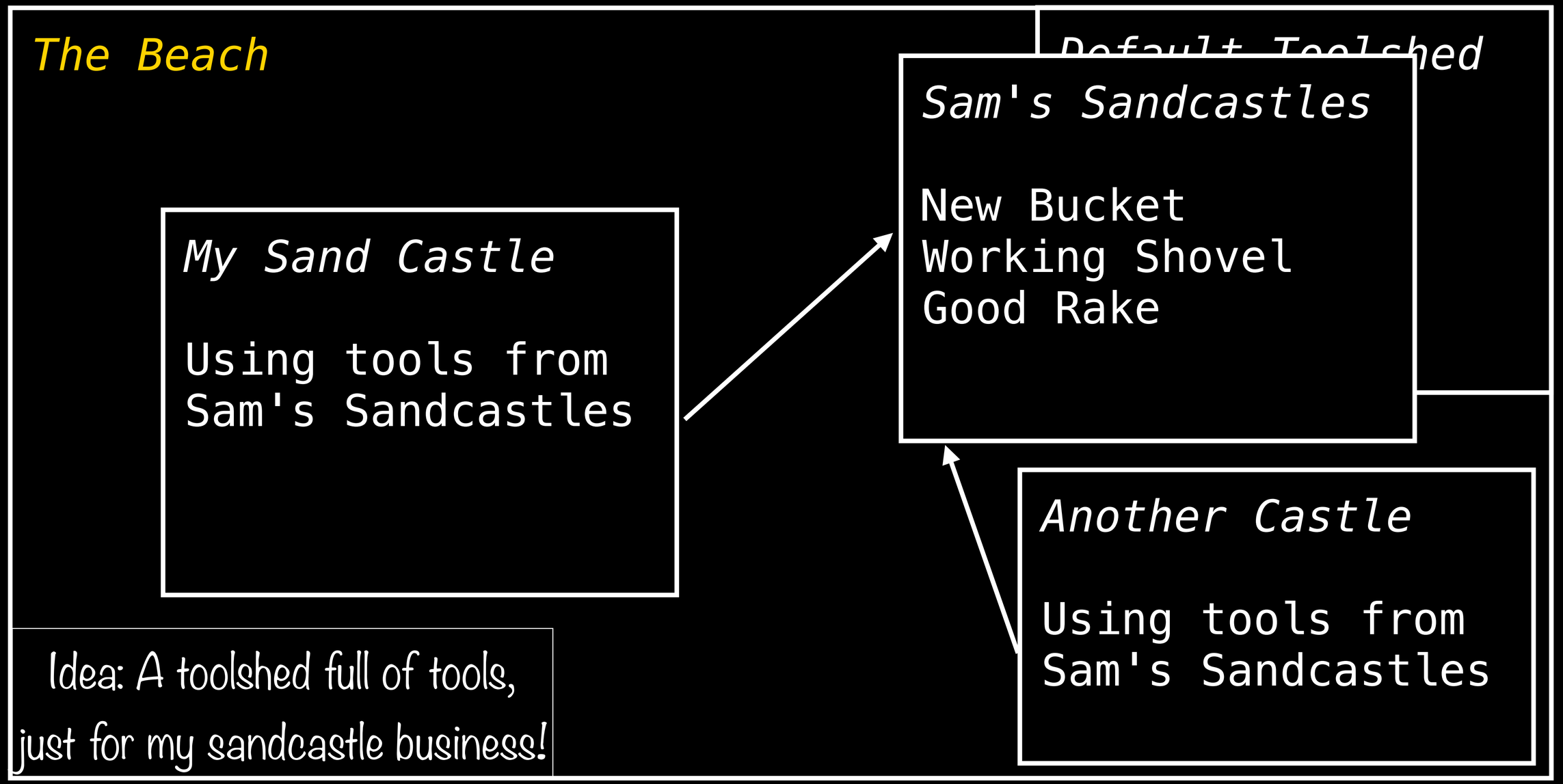
New Bucket
Working Shovel
Good Rake

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

An Analogy


The Beach
Default Toolshed
Sam's Sandcastles
New Bucket
Working Shovel
Good Rake
My Sand Castle
Using tools from
Sam's Sandcastles
Idea: A toolshed full of tools,
just for my sandcastle business!

An Analogy


The Beach
Default Toolshed
Sam's Sandcastles
New Bucket
Working Shovel
Good Rake
My Sand Castle
Using tools from
Sam's Sandcastles
Another Castle
Using tools from
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

Global Packages
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

Global Packages

cs41-env

Python3
iPython
Updated 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

Global Packages

cs41-env

Python3
iPython
Updated libraries

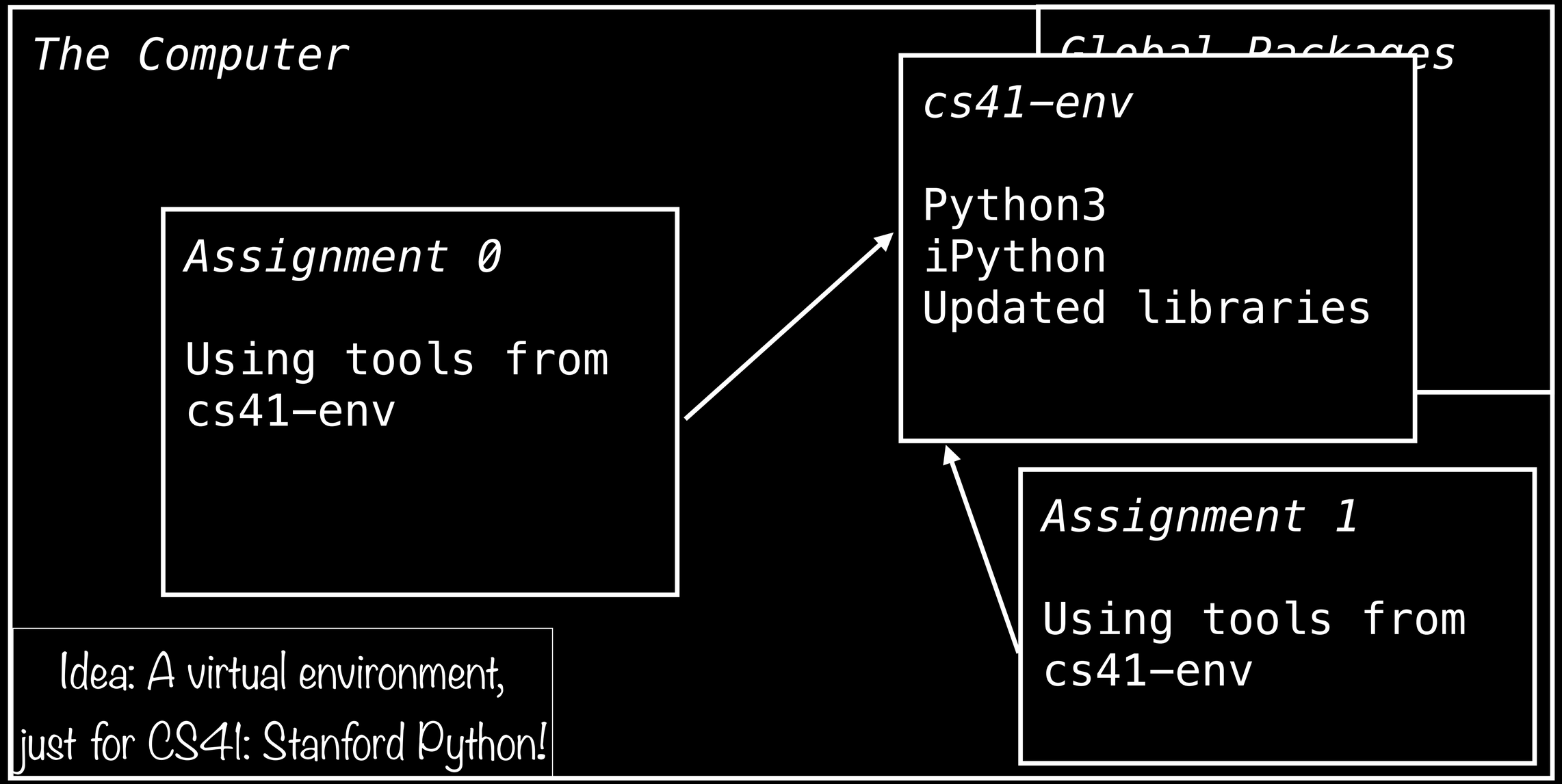
Assignment 0

Using tools from cs41-env

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...



A Final Question: How to Get New Tools?

A Final Question: How to Get New Tools?

`pip` is the preferred Python package manager. Use `pip`!

```
pip install numpy
```

A Final Question: How to Get New Tools?

`pip` is the preferred Python package manager. Use `pip`!

```
pip install numpy
```

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 return to the beach, you must enable some active toolshed.

Some Advice

Each time you return to the beach, you must enable some active toolshed.

Maintaining tools you control can be easier than maintaining global tools.

Some Advice

Each time you return to the beach, you must enable some active toolshed.

Maintaining tools you control can be easier than maintaining global tools.

Make sure you're using the right tools before building.

High Level: Setting Up the Toolshed

High Level: Setting Up the Toolshed

Install Python 3.7.2

High Level: Setting Up the Toolshed

Install Python 3.7.2

Create a virtual environment using Python 3.7.2

High Level: Setting Up the Toolshed

Install Python 3.7.2

Create a virtual environment using Python 3.7.2

Learn to activate and deactivate the virtual environment.

High Level: Setting Up the Toolshed

Install Python 3.7.2

Create a virtual environment using Python 3.7.2

Learn to activate and deactivate the virtual environment.

Install and upgrade packages in the virtual environment.

High Level: Setting Up the Toolshed

Install Python 3.7.2

Create a virtual environment using Python 3.7.2

Learn to activate and deactivate the virtual environment.

Install and upgrade packages in the virtual environment.

Optional: Use `virtualenvwrapper` for managed environments.

High Level: Setting Up the Toolshed

Install Python 3.7.2

Create a virtual environment using Python 3.7.2

Learn to activate and deactivate the virtual environment.

Install and upgrade packages in the virtual environment.

Optional: Use `virtualenvwrapper` for managed environments.

Detailed instructions online!

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

Transition

Fewer syntax features, more Python tools and tricks

Week 2: Data Structures

Week 3: Functions

Week 4: Functional Programming

Week 5: Object-Oriented Programming

Buffer Time: Setup + Lab!

Lab

Time to Experiment!

Make Friends!

CS41 Playlist

Today

Python Installations and Lab Setup

Handouts available online

Write some code!