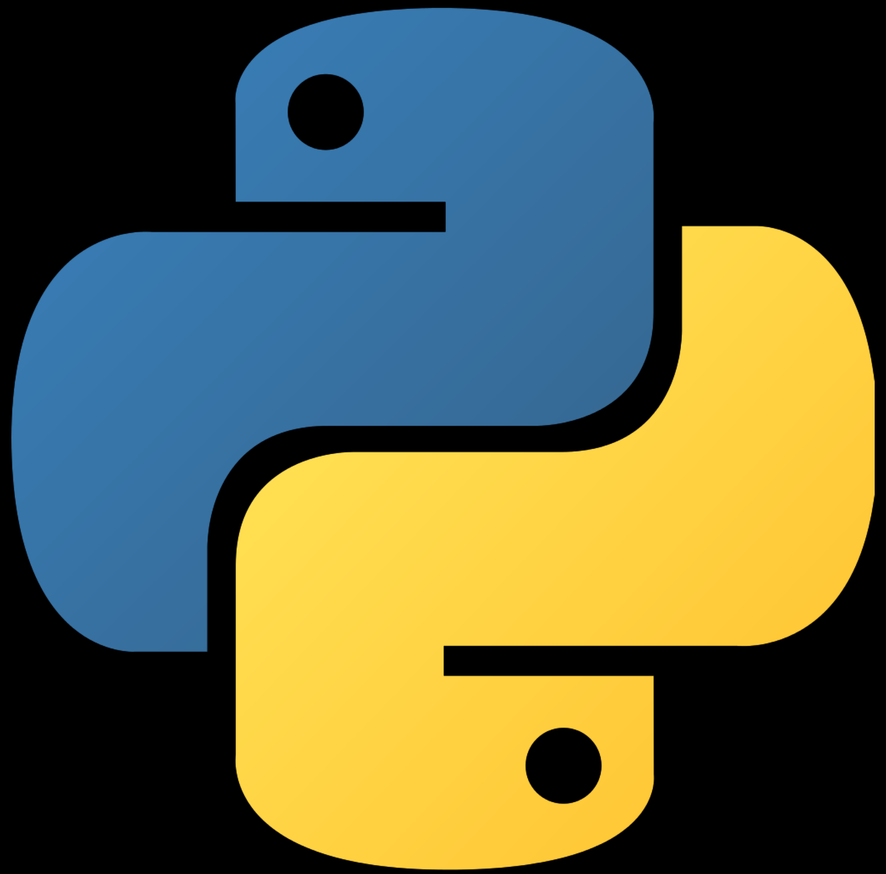
Welcome to Python!

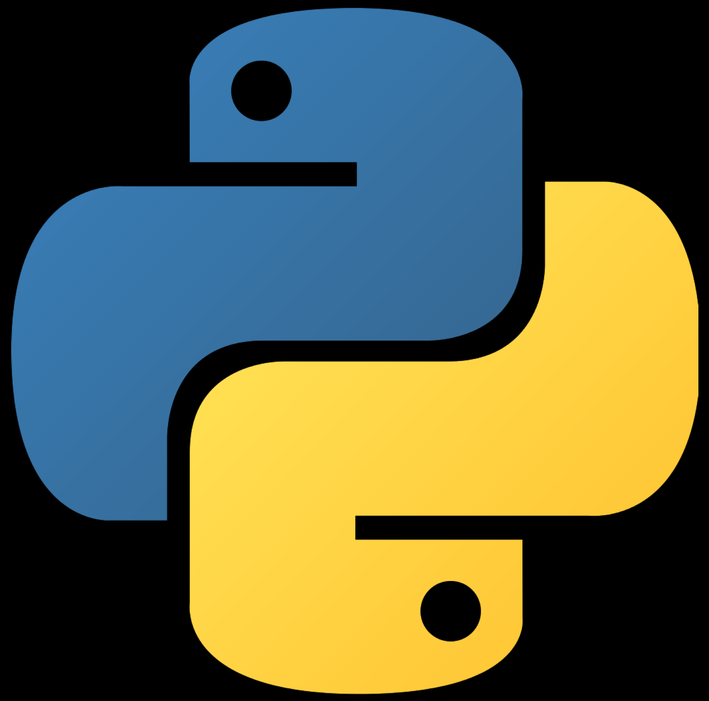
CS 41: hap.py code

The Python Programming Language



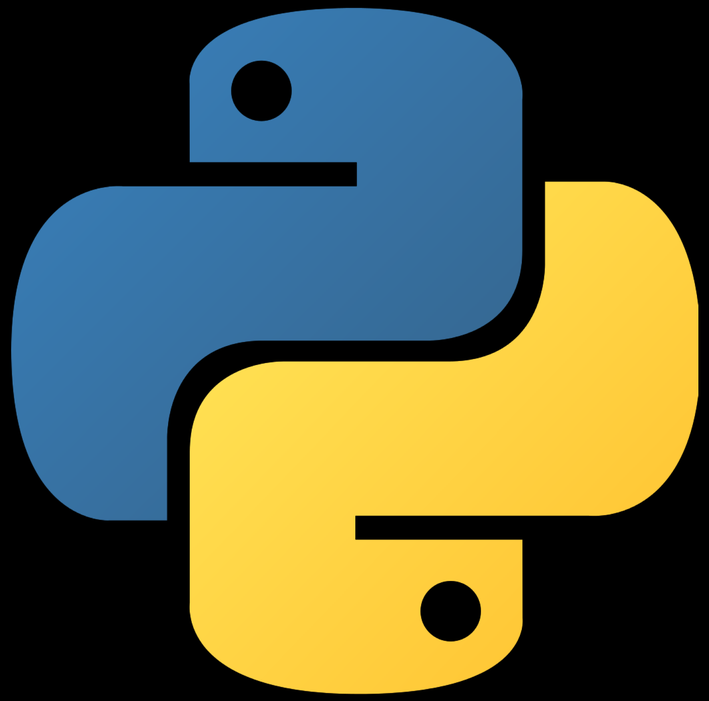
Agenda

Agenda



Welcome!

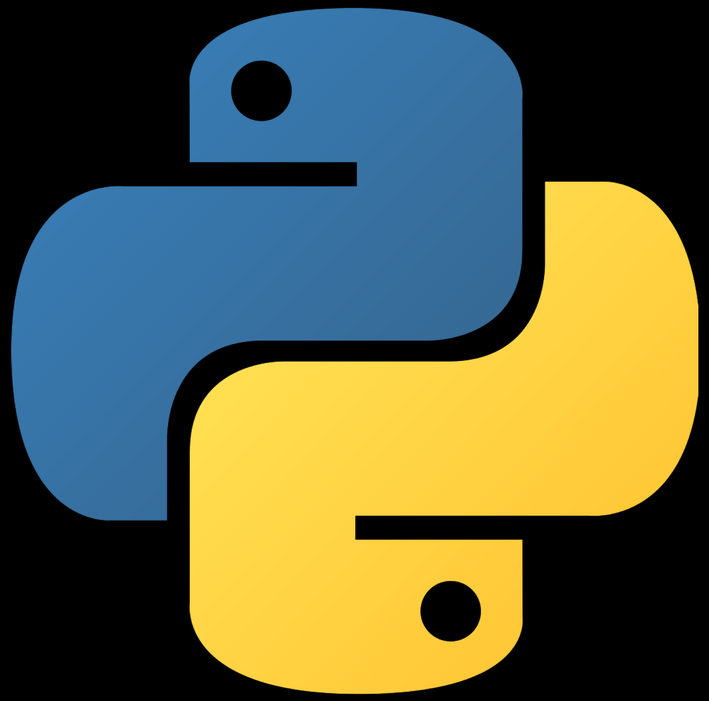
Agenda



Welcome!

Why Take CS41?

Agenda

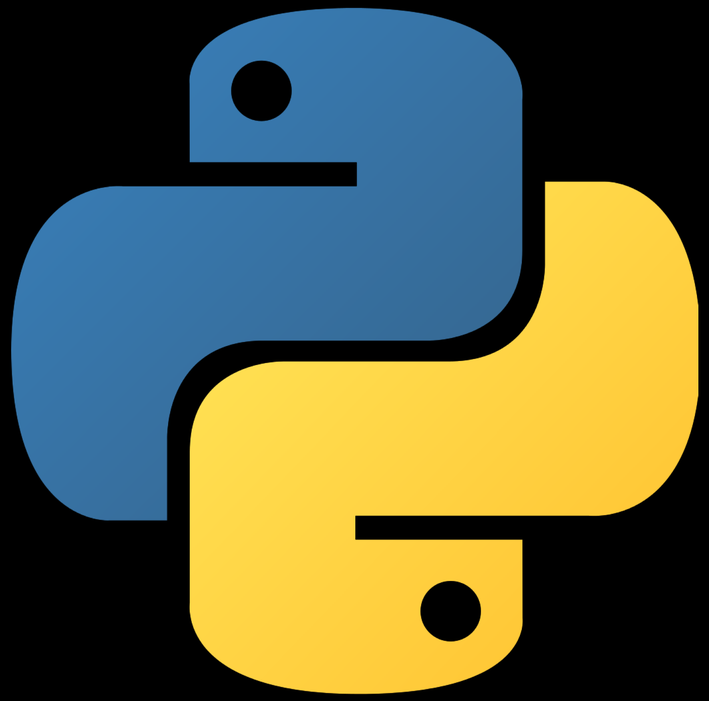


Welcome!

Why Take CS41?

What is Python?

Agenda



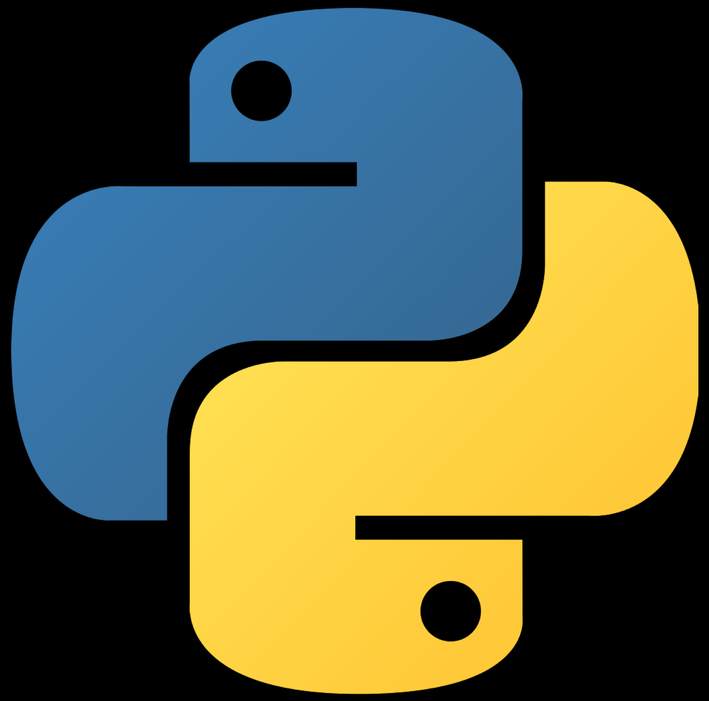
Welcome!

Why Take CS41?

What is Python?

Logistics

Agenda



Welcome!

Why Take CS41?

What is Python?

Logistics

Python Crash Course

Instructor

Sam Redmond

sredmond@stanford.edu

Course Helpers

Course Helpers

Joy Hsu

Divya Saini

Shrey Gupta

Andrew Kondrich

Christina Ramsey

Brahm Capoor

Emily Cohen

Norah Borus

Colin Kincaid

Ali Malik

Course Helpers

Joy Hsu

Divya Saini

Shrey Gupta

Andrew Kondrich

Christina Ramsey

Brahm Capoor

Emily Cohen

Norah Borus

Colin Kincaid

Ali Malik

staff@stanfordpython.com

You

You

International Relations

Chinese

Business

Public Policy

Computer Science

Physics

Education

Computational Biology

Biomedical Computation

Environmental Engineering

Classics

Mathematics

Law

Psychology

Medicine

Asian American Studies

Chemistry

History

Management Science & Engineering

Mechanical Engineering

Geophysics

Neuroscience

Economics

Aero/Astro

Symbolic Systems

Finance

English

Philosophy

Mathematical & Computational Science

Energy Resources Engineering

Bioengineering

Music

Linguistics

Biomedical Informatics

Electrical Engineering

Statistics

East Asian Studies

Science, Technology and Society

Product Design

Art History

Why CS41?

Course Goals

Course Goals

1. Develop skills with Python fundamentals, both old and new

Course Goals

1. Develop skills with Python fundamentals, both old and new
2. Learn to recognize and write "good" Python

Course Goals

1. Develop skills with Python fundamentals, both old and new
2. Learn to recognize and write "good" Python
3. Gain experience with practical Python tasks

Course Goals

1. Develop skills with Python fundamentals, both old and new
2. Learn to recognize and write "good" Python
3. Gain experience with practical Python tasks
4. Understand Python's strengths (and weaknesses)

Course Goals

1. Develop skills with Python fundamentals, both old and new
2. Learn to recognize and write "good" Python
3. Gain experience with practical Python tasks
4. Understand Python's strengths (and weaknesses)

Questions

Questions

What is Python?

Questions

What is Python?

Why Python?

Questions

What is Python?

Why Python?

Will Python help me get a job?

History of Python

History of Python

Guido van Rossum
BDFL

History of Python

Guido van Rossum
BDFL

Python 1: 1994

History of Python

Guido van Rossum
BDFL

Python 1: 1994

Python 2: 2000

History of Python

Guido van Rossum
BDFL

Python 1: 1994

Python 2: 2000

Python 3: 2008

History of Python

Guido van Rossum
BDFL

Python 1: 1994

Python 2: 2000

Python 3: 2008

Specifically, we're using Python 3.7.2

Philosophy of Python

```
>>> import this
```

```
>>> import this

The Zen of Python, by Tim Peters
```

```
>>> import this

The Zen of Python, by Tim Peters
Beautiful is better than ugly.
```

```
>>> import this

The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
```

```
>>> import this

The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
```

```
>>> import this

The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
```

```
>>> import this

The Zen of Python, by Tim Peters
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.
```

```
>>> import this

The Zen of Python, by Tim Peters
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.
```

```
>>> import this

The Zen of Python, by Tim Peters
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.
```

>>> import this

>>> import this

```
Special cases aren't special enough to break the rules.
```

>>> import this

```
Special cases aren't special enough to break the rules.
Although practicality beats purity.
```

>>> import this

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

>>> import this

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

>>> import this

```
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
```

>>> import this

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

>>> import this

```
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
```

>>> import this

Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.

>>> import this

```
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
```

>>> import this

```
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
```

>>> import this

```
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
```

>>> import this

```
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Programmers are more important than programs

“Hello World” in Java

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

“Hello World” in C++

```
#include <iostream>
using namespace std;

int main() {
    cout << "Hello World!" << endl;
}
```

Double Yuck

“Hello World” in Python

```
print("Hello world!")
```

Who Uses Python?

Python at Stanford

Python at Stanford

CEE 345: Network Analysis for Urban Systems

COMM 177P: Programming in Journalism

COMM 382: Big Data and Causal Inference

CS 375: Large-Scale Neural Network Modeling for Neuroscience

GENE 211: Genomics

LINGUIST 276: Quantitative Methods in Linguistics

MI 245: Computational Modeling of Microbial Communities

MS&E 448: Big Financial Data and Algorithmic Trading

PHYSICS 368: Computational Cosmology and Astrophysics

PSYCH 162: Brain Networks

STATS 155: Statistical Methods in Computational Genetics

Python in Business

Python in Business

Dropbox

yelp

Eventbrite

venmo

Quora

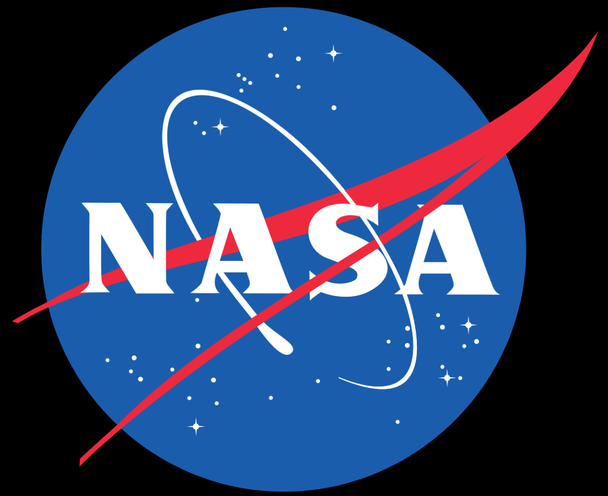
Google

Instagram

YouTube

Other Python Users

Other Python Users



5-Minute Break

Logistics

Logistics

Logistics

Lectures Tue / Thu, 4:30-5:50, 380-380D

Logistics

Lectures Tue / Thu, 4:30-5:50, 380-380D

Units 2 CR/NC

Logistics

Lectures Tue / Thu, 4:30-5:50, 380-380D

Units 2 CR/NC

Website stanfordpython.com

Logistics

Lectures Tue / Thu, 4:30-5:50, 380-380D

Units 2 CR/NC

Website stanfordpython.com

Bookmark it! We'll post announcements, lecture slides, and handouts online.

Logistics

Lectures Tue / Thu, 4:30-5:50, 380-380D

Units 2 CR/NC

Website stanfordpython.com

Prereqs CS106B/X or equivalent

Bookmark it! We'll post announcements, lecture slides, and handouts online.

Logistics

Lectures Tue / Thu, 4:30-5:50, 380-380D

Units 2 CR/NC

Website stanfordpython.com

Prereqs CS106B/X or equivalent

Enrollment Cap 40 :(

Bookmark it! We'll post announcements, lecture slides, and handouts online.

Logistics

Logistics

Attendance Required. At most 2 unexcused absences.

Logistics

iamhere.stanfordpython.com

Attendance Required. At most 2 unexcused absences.

Logistics

iamhere.stanfordpython.com

Attendance Required. At most 2 unexcused absences.

Auditing Encouraged

Logistics

iamhere.stanfordpython.com

Attendance Required. At most 2 unexcused absences.

Auditing Encouraged

Waitlist Rolling

Logistics

iamhere.stanfordpython.com

Attendance Required. At most 2 unexcused absences.

Auditing Encouraged

Waitlist Rolling

Piazza Sign up!

Logistics

Logistics

Assignments 4 in total

Logistics

Assignments 4 in total

Grading Functionality and style, on a checkmark scale

Logistics

Assignments 4 in total

Grading Functionality and style, on a checkmark scale

Credit For both functionality and style, average a check

Logistics

Assignments 4 in total

Grading Functionality and style, on a checkmark scale

Credit For both functionality and style, average a check

Late Days Two 24-hour extensions

Logistics

Assignments 4 in total

Grading Functionality and style, on a checkmark scale

Credit For both functionality and style, average a check

Late Days Two 24-hour extensions

Honor Code Don’t cheat

Logistics

Assignments 4 in total

Grading Functionality and style, on a checkmark scale

Credit For both functionality and style, average a check

Late Days Two 24-hour extensions

Honor Code Don't cheat

More specifics can be found on the Course Info handout

The Big Picture

The Road Ahead - The Python Language

The Road Ahead - The Python Language

Week 1 Python Fundamentals

Week 2 Data Structures

Week 3 Functions

Week 4 Functional Programming

Week 5 Object-Oriented Python

The Road Ahead - Python Tools

The Road Ahead - Python Tools

Week 6 Standard Library

Week 7 Third-Party Tools

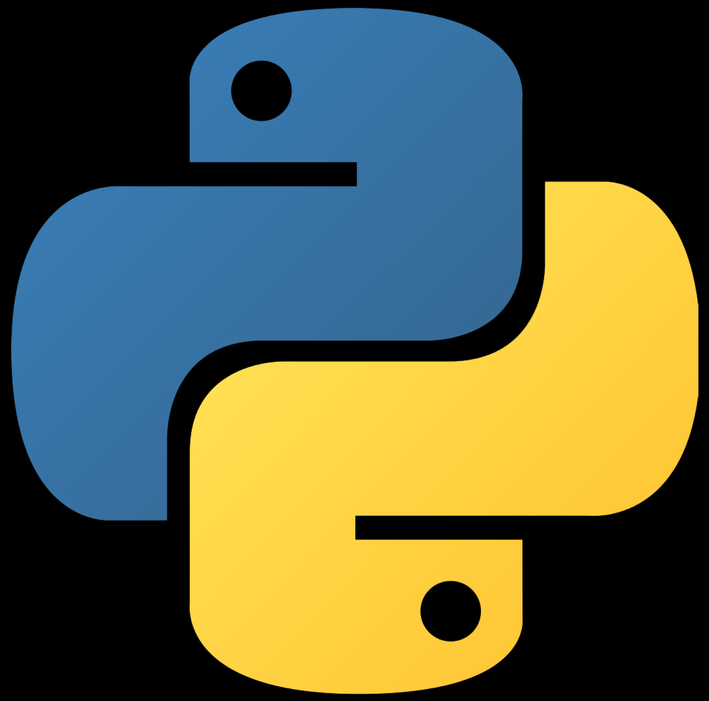
Week 8 Ecosystem

Week 9 Advanced Topics

Week 10 Projects!

Let’s Get Started!

Python Basics



- Interactive Interpreter
- Comments
- Variables and Types
- Numbers and Booleans
- Strings and Lists
- Console I/O
- Control Flow
- Loops
- Functions

Interactive Interpreter

```
sredmond$
```

Interactive Interpreter

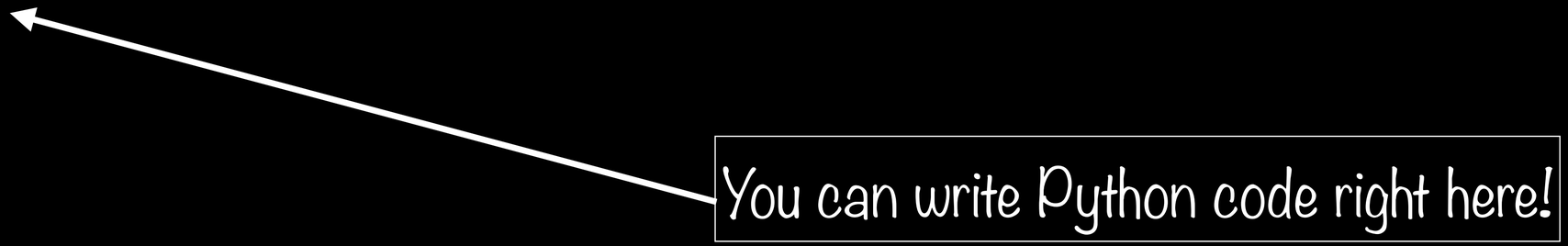
```
sredmond$ python3
```

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

```
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.
>>>
```



A Big Deal

A Big Deal

Immediate gratification!

A Big Deal

Immediate gratification!

Sandboxed environment to experiment with Python

A Big Deal

Immediate gratification!

Sandboxed environment to experiment with Python

Shortens code-test-debug cycle to seconds

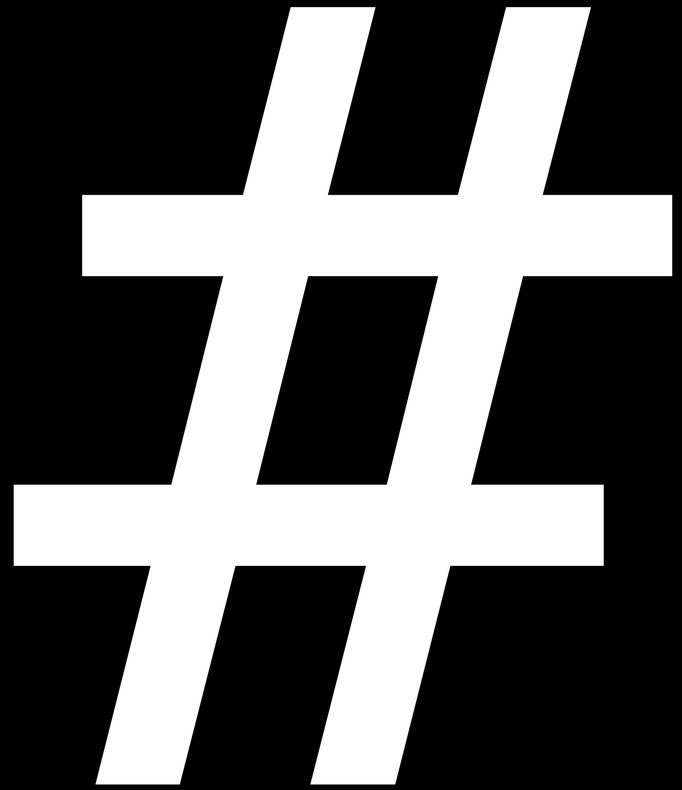
A Big Deal

Immediate gratification!

Sandboxed environment to experiment with Python

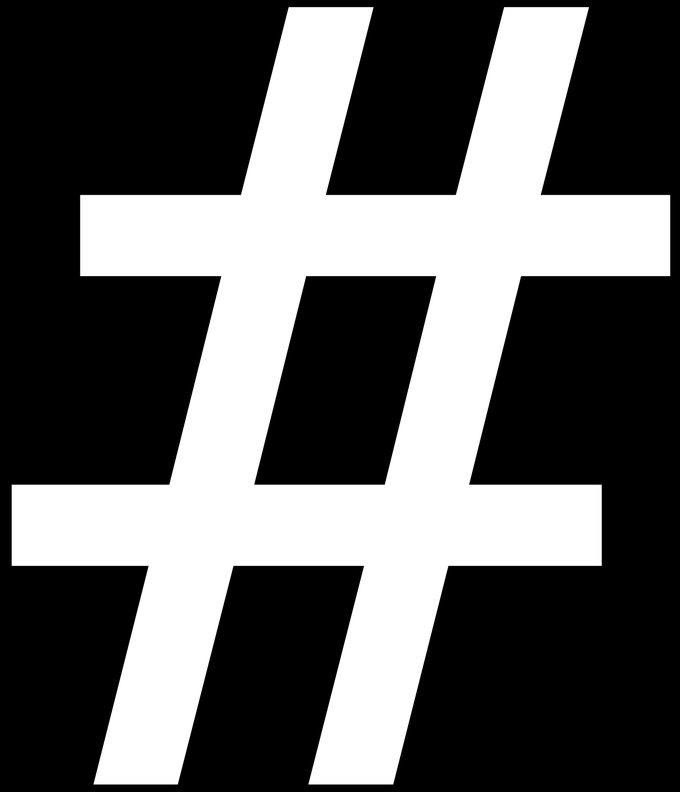
Shortens code-test-debug cycle to seconds

The interactive interpreter is your new best friend



Hashtag

Pound Sign



Sharp

Number Sign

Octothorpe

Comments

Comments

```
# Single line comments start with a '#'
```

Comments

```
# Single line comments start with a '#'

"""
Multiline comments can be written between
three "s and are often used as function
and module comments.
"""
```

Variables

Variables

Variables

```
x = 2
```

No semicolon!

Variables

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

No semicolon!

Variables

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

x = "Hello, I'm "
```

No semicolon!

Variables

No semicolon!

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

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

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?

In Java or C++

```
int x = 0;
```

Where's My Type?

In Python

```
x = 0
```

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

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

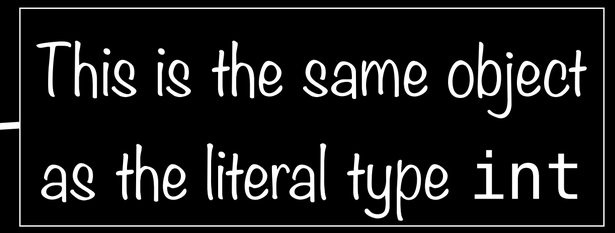
```
type(1)          # => <class 'int'>
type("Hello")    # => <class 'str'>
type(None)       # => <class 'NoneType'>
```

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'>
```



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

Python's dynamic type system is fascinating! More on Wed.

Numbers and Math

Numbers and Math

Numbers and Math

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

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

Numbers and Math

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

1 + 1           # => 2
8 - 1           # => 7
10 * 2          # => 20
5 / 2           # => 2.5
13 / 4          # => 3.25
9 / 3           # => 3.0
7 / 1.4         # => 5.0
```

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

Numbers and Math

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

1 + 1           # => 2
8 - 1           # => 7
10 * 2          # => 20
5 / 2           # => 2.5
13 / 4          # => 3.25
9 / 3           # => 3.0
7 / 1.4         # => 5.0

7 // 3          # => 2  (integer division)
7 % 3           # => 1  (integer modulus)
2 ** 4          # => 16 (exponentiation)
```

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

Booleans

Booleans

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

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

Booleans

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

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

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

Booleans

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

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

1 == 1              # => True
2 * 3 == 5          # => False
1 != 1              # => False
2 * 3 != 5          # => True
```

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

Booleans

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

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

1 == 1              # => True
2 * 3 == 5          # => False
1 != 1              # => False
2 * 3 != 5          # => True

1 < 10              # => True
2 >= 0              # => True
1 < 2 < 3           # => True  (1 < 2 and 2 < 3)
1 < 2 >= 3          # => False (1 < 2 and 2 >= 3)
```

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

Strings

Strings

Strings

No `char` in Python!

Both `'` and `"` create string literals

Strings

No `char` in Python!
Both ' and " create string literals

```
greeting = 'Hello'
group = "wørld"  # Unicode by default
```

Strings

No `char` in Python!
Both `'` and `"` create string literals

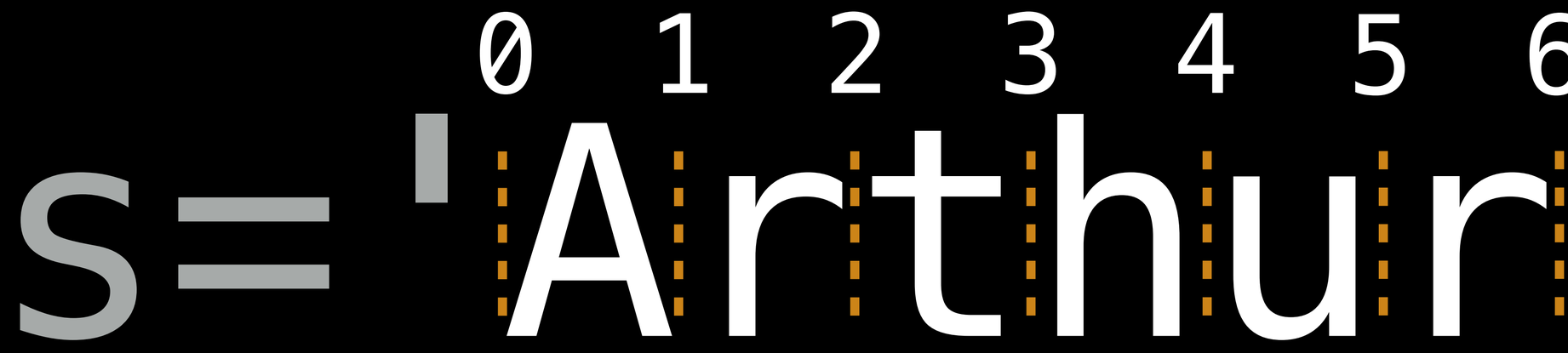
```
greeting = 'Hello'
group = "wørld"  # Unicode by default


greeting + ' ' + group + "!" # => 'Hello wørld!'
```

Indexing

```
  0 1 2 3 4 5 6
S='Arthur'
```

Indexing

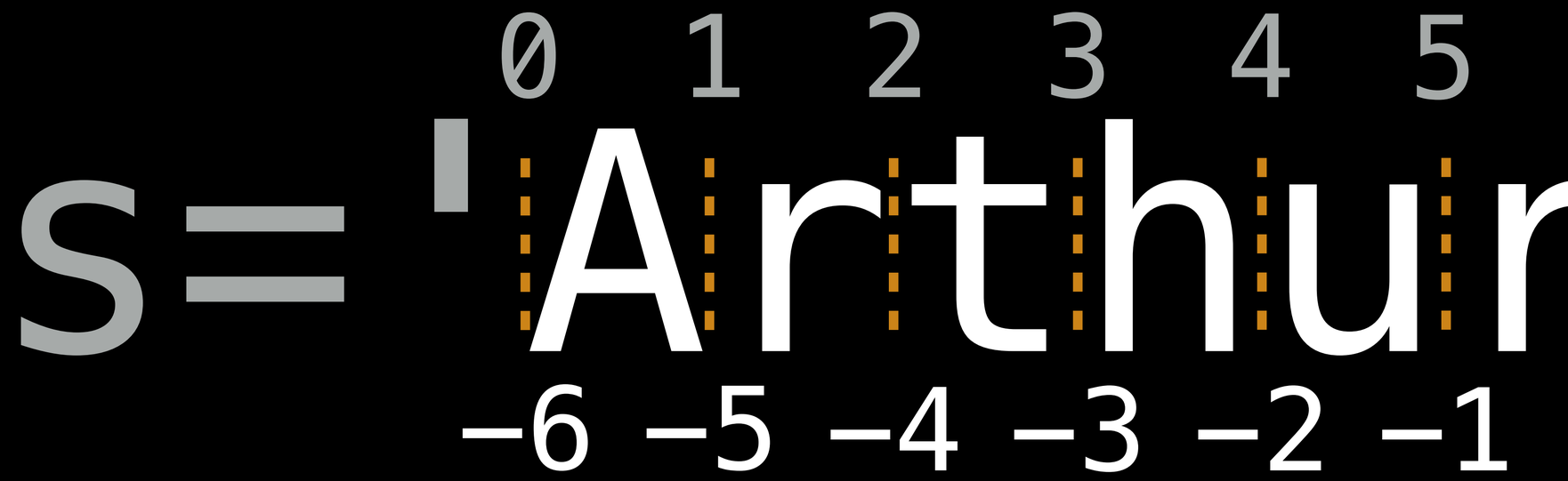


Indexing

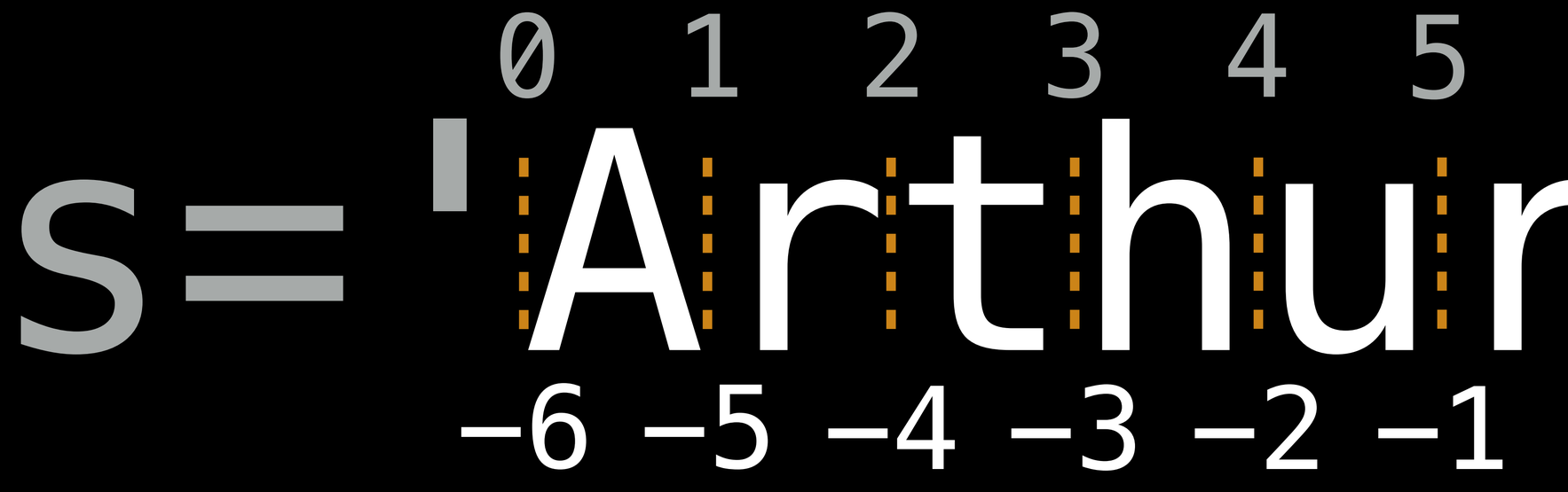
```
  0 1 2 3 4 5 6
s='Arthur'
```

```
s[0] == 'A'
s[1] == 'r'
s[4] == 'u'
s[6] # Bad!
```

Negative Indexing



Negative Indexing




```
s[-1] == 'r'
s[-2] == 'u'
s[-4] == 't'
s[-6] == 'A'
```

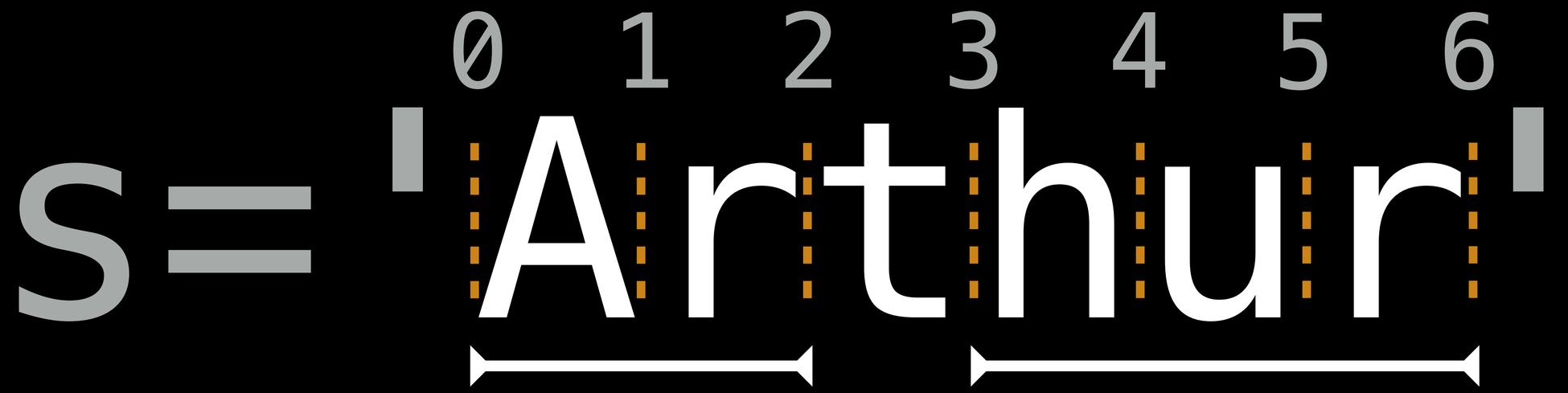
Slicing

Slicing

Slicing


```
s[0:2] ==  'Ar'
```

Slicing

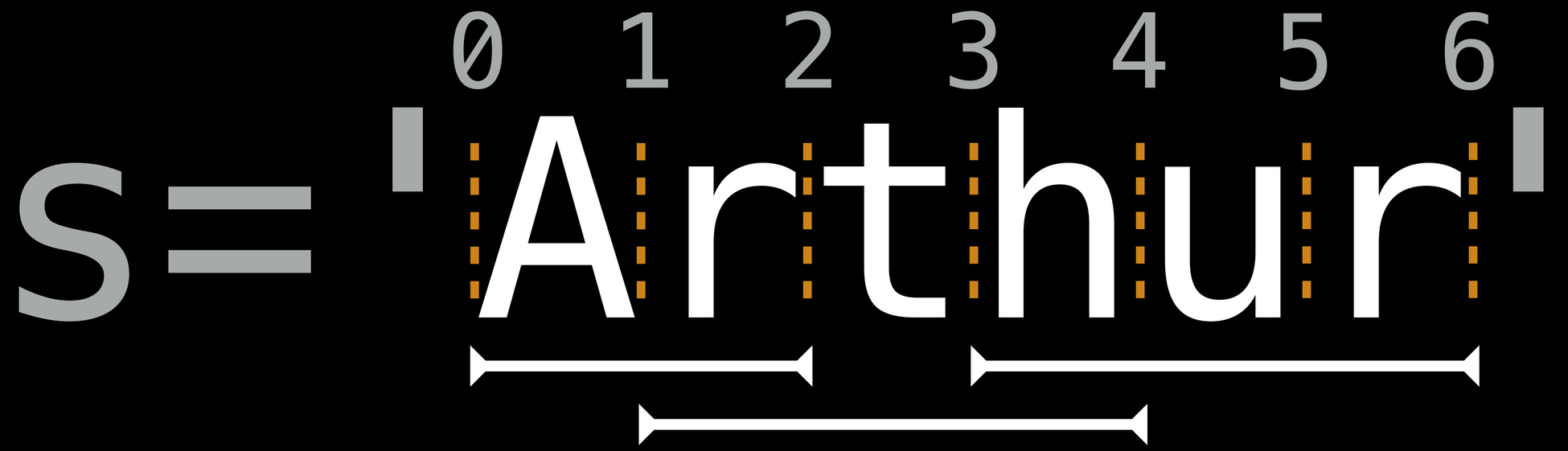



```
s[0:2] ==  'Ar'
```

Slicing


```
s[0:2] ==  'Ar'
s[3:6] == 'hur'
```

Slicing




```
s[0:2] ==  'Ar'
s[3:6] == 'hur'
s[1:4] == 'rth'
```

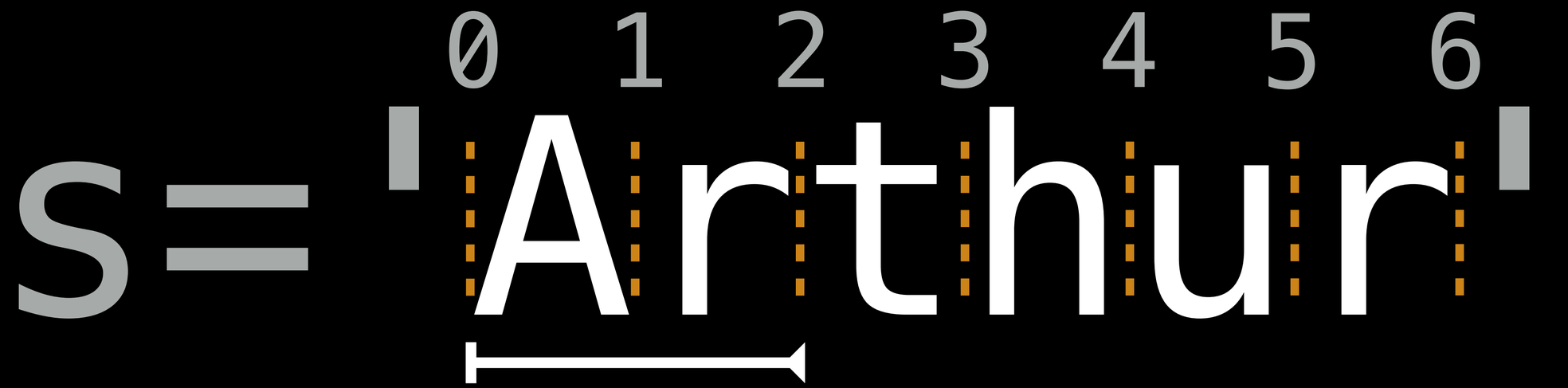
Strings

Implicitly starts at 0

Implicitly ends at the end

```
s[:2] ==  'Ar'
s[3:] == 'hur'
```

Strings

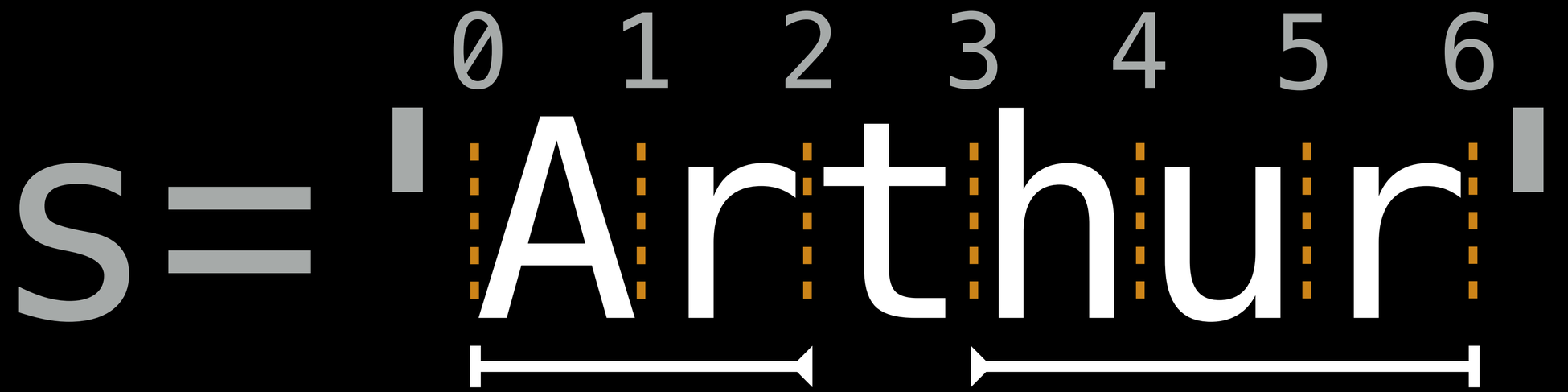


Implicitly starts at 0

```
s[:2] ==  'Ar'
s[3:] == 'hur'
```

Implicitly ends at the end

Strings



Implicitly starts at 0

```
s[:2] ==  'Ar'
s[3:] == 'hur'
```

Implicitly ends at the end

Strings

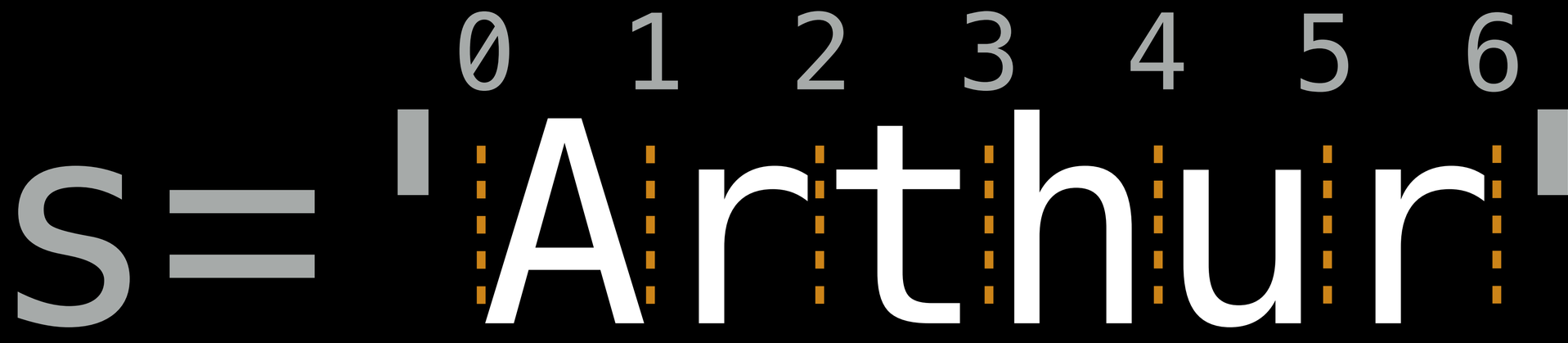
Strings

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

Can also pass a step size

```
s[1:5:2] == 'rh'
s[4::-2] == 'utA'
```

Strings



Can also pass a step size

```
s[1:5:2] == 'rh'
s[4::-2] == 'utA'
 s[::-1] == 'ruhtrA'
```

One way to reverse a string

Converting Values

Converting Values

```
str(42)          # => "42"
```

All objects have a string representation

Converting Values

```
str(42)          # => "42"
int("42")        # => 42
```

All objects have a string representation

Converting Values

```
str(42)          # => "42"
int("42")        # => 42
float("2.5")     # => 2.5
```

All objects have a string representation

Converting Values

```
str(42)          # => "42"
int("42")        # => 42
float("2.5")     # => 2.5
float("1")       # => 1.0
```

All objects have a string representation

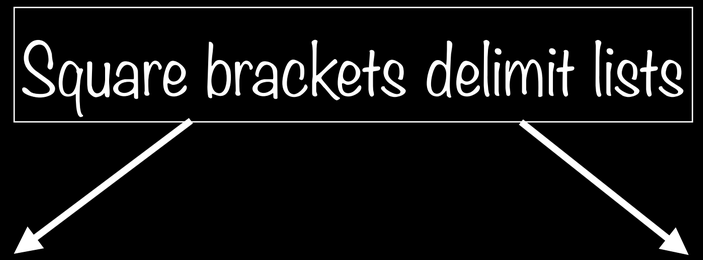
Lists

Dive into Python data structures Week 2!

Lists

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

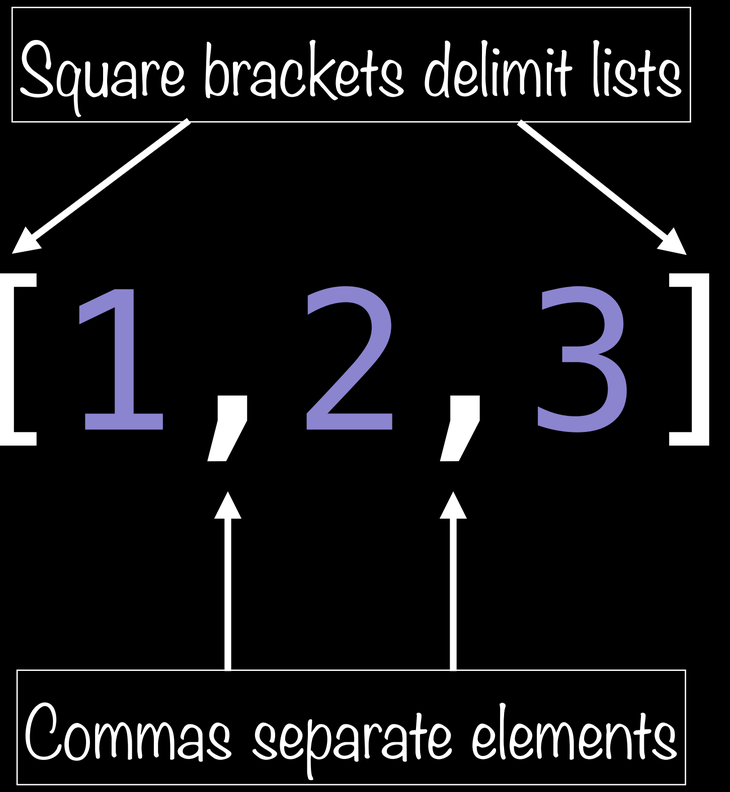
Lists



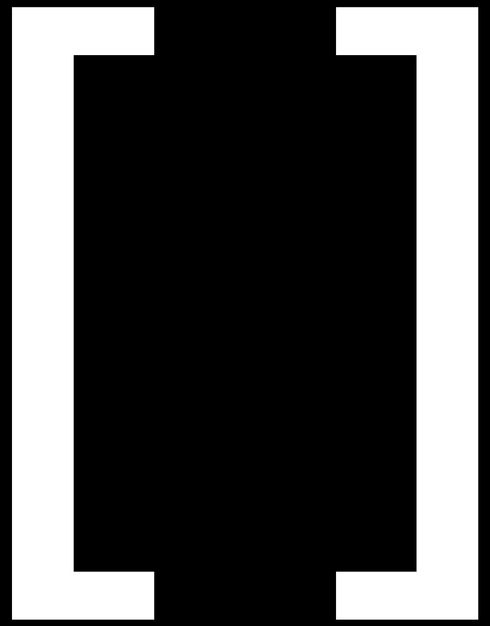

```
easy_as = [1,2,3]
```

Lists

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



Lists



Versatile

Incredibly common

≈ ArrayList / Vector

Basic Lists

Basic Lists

```
# Create a new list
empty = []
letters = ['a', 'b', 'c', 'd']
numbers = [2, 3, 5]
```

Basic Lists

```
# Create a new list
empty = []
letters = ['a', 'b', 'c', 'd']
numbers = [2, 3, 5]

# Lists can contain elements of different types
mixed = [4, 5, "seconds"]
```

Basic Lists

```
# Create a new list
empty = []
letters = ['a', 'b', 'c', 'd']
numbers = [2, 3, 5]

# Lists can contain elements of different types
mixed = [4, 5, "seconds"]

# Append elements to the end of a list
numbers.append(7)   # numbers == [2, 3, 5, 7]
numbers.append(11)  # numbers == [2, 3, 5, 7, 11]
```

Inspecting List Elements

Inspecting List Elements

```
letters = ['a', 'b', 'c', 'd']
numbers = [2, 3, 5, 7, 11]
```

Inspecting List Elements

```
letters = ['a', 'b', 'c', 'd']
numbers = [2, 3, 5, 7, 11]

# Access elements at a particular index
numbers[0]  # => 2
numbers[-1]  # => 11
```

Inspecting List Elements

```
letters = ['a', 'b', 'c', 'd']
numbers = [2, 3, 5, 7, 11]

# Access elements at a particular index
numbers[0]  # => 2
numbers[-1]  # => 11

# You can also slice lists - the same rules apply
letters[:3]  # => ['a', 'b', 'c']
numbers[1:-1]  # => [3, 5, 7]
```

Nested Lists

Nested Lists

```
# Lists really can contain anything – even other lists!
combo = [letters, numbers]
combo  # => [['a', 'b', 'c', 'd'], [2, 3, 5, 7, 11]]
```

Nested Lists

```
# Lists really can contain anything – even other lists!
combo = [letters, numbers]
combo  # => [['a', 'b', 'c', 'd'], [2, 3, 5, 7, 11]]
combo[0]  # => ['a', 'b', 'c', 'd']
```

Nested Lists

```
# Lists really can contain anything – even other lists!
combo = [letters, numbers]
combo  # => [['a', 'b', 'c', 'd'], [2, 3, 5, 7, 11]]
combo[0]  # => ['a', 'b', 'c', 'd']
combo[0][1]  # => 'b'
```

Nested Lists

```
# Lists really can contain anything – even other lists!
combo = [letters, numbers]
combo  # => [['a', 'b', 'c', 'd'], [2, 3, 5, 7, 11]]
combo[0]  # => ['a', 'b', 'c', 'd']
combo[0][1]  # => 'b'
combo[1][2:]  # => [5, 7, 11]
```

General Queries

General Queries

```
# Length (len)
len([])  # => 0
len("python")  # => 6
len([4, 5, "seconds"])  # => 3
```

General Queries

```
# Length (len)
len([])  # => 0
len("python")  # => 6
len([4, 5, "seconds"])  # => 3


# Membership (in)
0 in []  # => False
```

General Queries

```
# Length (len)
len([])  # => 0
len("python")  # => 6
len([4, 5, "seconds"])  # => 3


# Membership (in)
0 in []  # => False
'y' in "python"  # => True
```

General Queries

```
# Length (len)
len([])  # => 0
len("python")  # => 6
len([4, 5, "seconds"])  # => 3


# Membership (in)
0 in []  # => False
'y' in "python"  # => True
"minutes" in [4, 5, "seconds"]  # => False
```

Console I/O

Console I/O

Console I/O

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

`input` prompts the user for input

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?
```

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
```

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
```

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

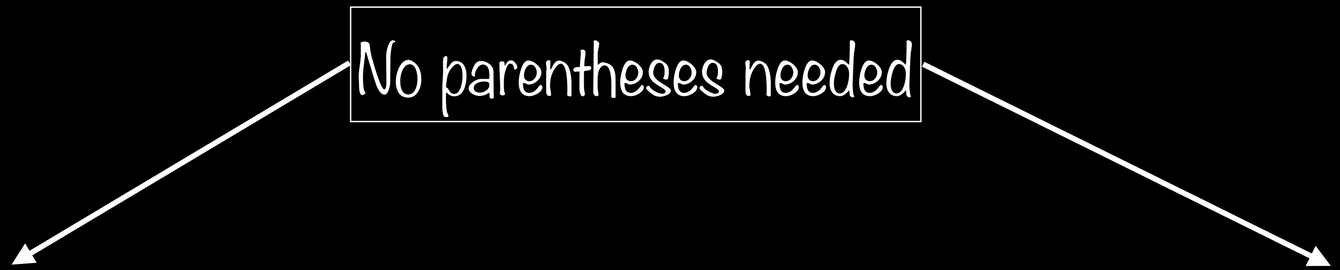
`print` can be used in many different ways!

Control Flow

If Statements

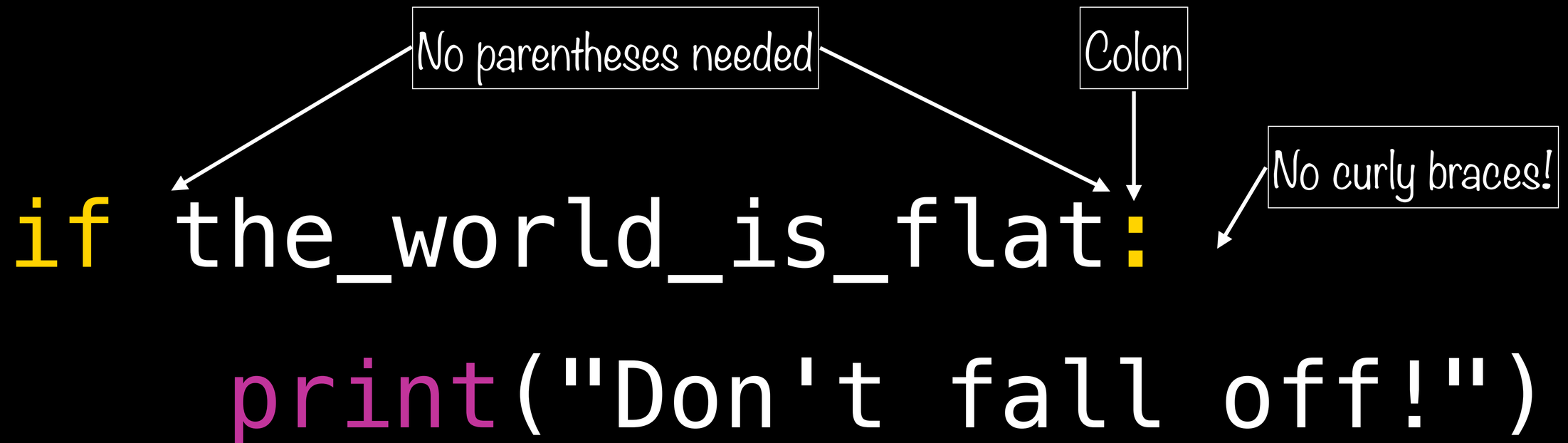
```
if the_world_is_flat:
    print("Don't fall off!")
```

If Statements



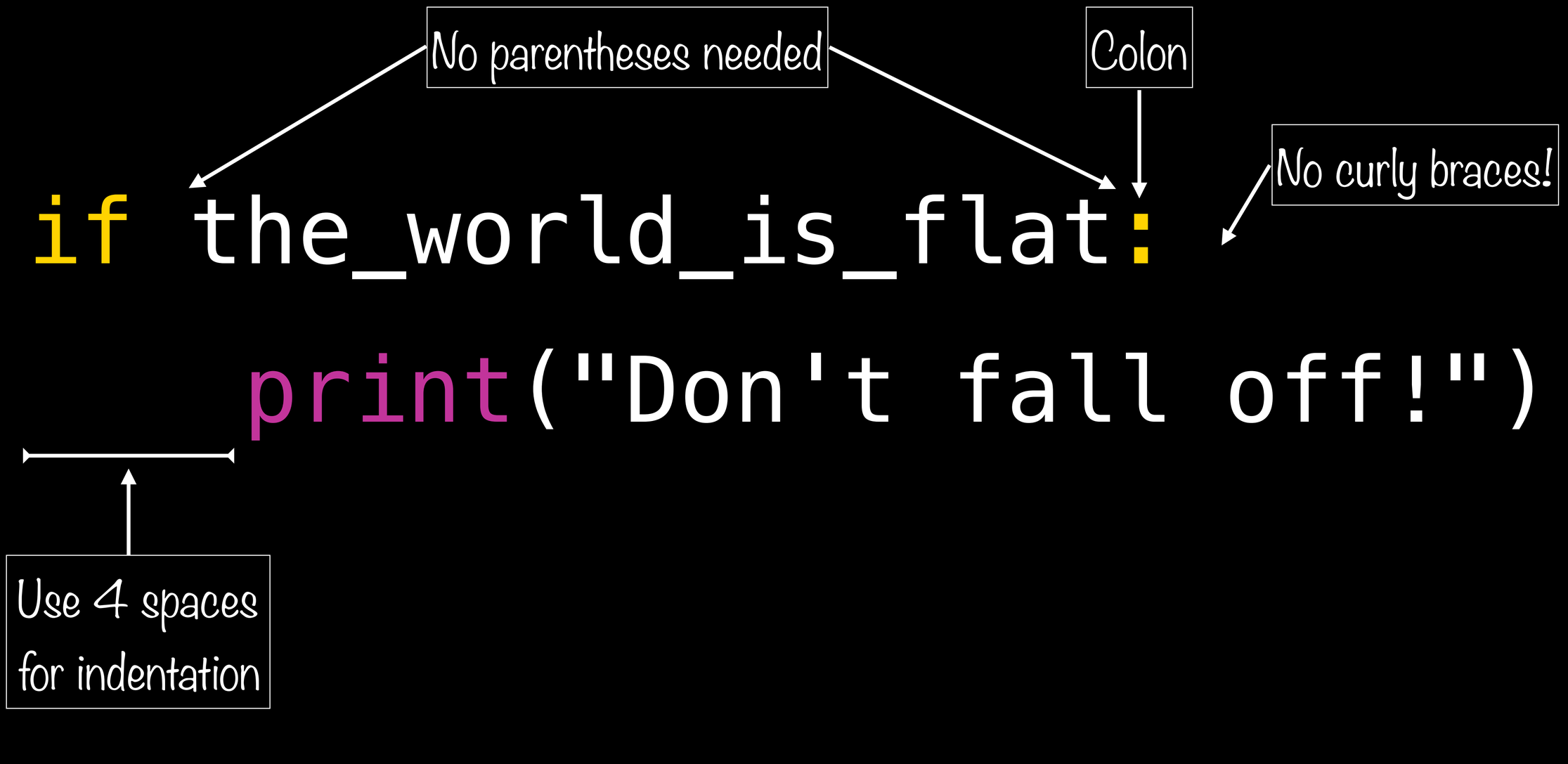

```
if the_world_is_flat:
    print("Don't fall off!")
```

If Statements




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

If Statements



4 Spaces?! No Braces?!

Zen of Python

Readability counts

Can be configured in most development environments

`elif` and `else`

```
if some_condition:
    print("Some condition holds")
elif other_condition:
    print("Other condition holds")
else:
    print("Neither condition holds")
```

zero or more `elif`s

`else` is optional

Python has no `switch` statement, opting for `if/elif/else` chains

Palindrome?

Spelled the same backwards and forwards

Palindrome?

Spelled the same backwards and forwards

```
# Is a user-submitted word a palindrome?
word = input("Please enter a word: ")
reversed_word = word[::-1]
```

Palindrome?

Spelled the same backwards and forwards

```
# Is a user-submitted word a palindrome?
word = input("Please enter a word: ")
reversed_word = word[::-1]
```

Pause: How did this work again?

Palindrome?

Spelled the same backwards and forwards

```
# Is a user-submitted word a palindrome?
word = input("Please enter a word: ")
reversed_word = word[::-1]
if word == reversed_word:
    print("Hooray! You entered a palindrome")
```

Pause: How did this work again?

Palindrome?

Spelled the same backwards and forwards

```
# Is a user-submitted word a palindrome?
word = input("Please enter a word: ")
reversed_word = word[::-1]
if word == reversed_word:
    print("Hooray! You entered a palindrome")
else:
    print("You did not enter a palindrome")
```

Pause: How did this work again?

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)
else:
    print("There's no data!")
```

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!")
```

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=', ')
# => 2, 3, 4, 5,
```

`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='*')
# => S*T*A*R*
```

`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='*')
# => 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

n = 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

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

Prime Number Generator

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

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

More to See

More to See

Keyword Arguments

Variadic Argument Lists

Default Argument Values

Unpacking Arguments

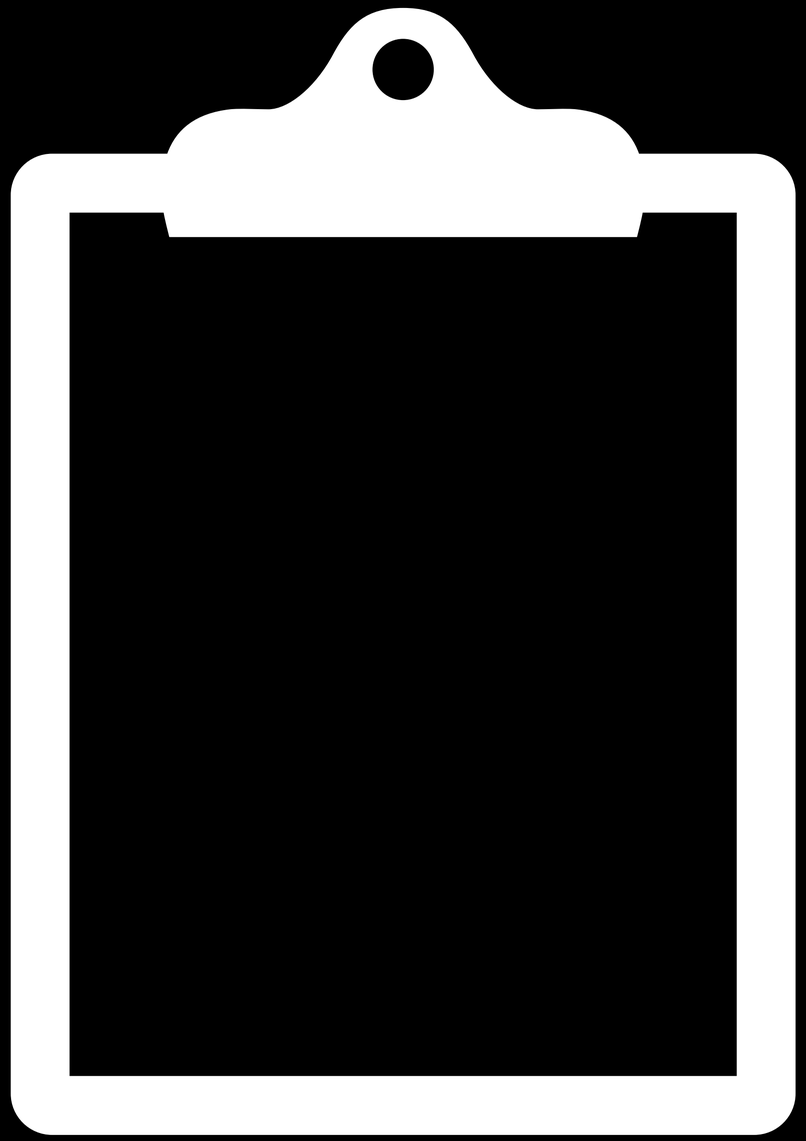
Anonymous Functions

First-Class Functions

Functional Programming

Next Time

More Python Fundamentals!



Types and Objects

String Formatting

File I/O

Using Scripts

Configuring Python 3

Lab!

Appendix

Citations

Examples in slides and interactive activities in this course are drawn, with or without modification, from:

http://learnpythonthehardway.org/

http://learnxinyminutes.com/docs/python3/

https://docs.python.org/3/tutorial/index.html