Welcome to Python!

Python 4 LM : Hap.py Code
The Python Programming Language

Welcome to Python 4 LM

Today in Python for LM

Welcome to Pyton 4 LM!

Today Python for LM

- Welcome!
- Who are we?
- Who are you?
- Why take this course
- What is Python?
- Course Logistics
- Python Crash Course

Who are we?
INSTRUCTORS

Abdallah Khemais
abdallah.khemais@gmail.com
khemais.abdallah@isitc.u-sousse.tn

Who are we?
COURSE STAFF

?

Our thanks to…

CONTRIBUTORS THROUGH THE YEARS

Who are you?

Mechanical Engineering

Systems Engineering

Electrical Engineering

Computer Science

Earth Systems

Biophysics

History

Management Science and Engineering

International Relations

Bioengineering

Civil and Environmental Engineering

Medicine

Linguistics

Mathematical and Computational Science

Symbolic Systems

Economics

Political Economics

Science and Technology for Society

Sociophonetics

Science and Technology in Society

Computational Biology

Why Python ?

Course Goals

Why Python ?

Course Goals

1. Python fundamentals

Why Python ?

Course Goals

1. Python fundamentals
2. Recognize and write “good” Python

Why Python ?

Course Goals

1. Python fundamentals
2. Recognize and write “good” Python
3. Use Python for practical tasks…
4. …especially machine learning and data science

Why Python ?

Course Goals

1. Python fundamentals
2. Recognize and write “good” Python
3. Use Python for practical tasks…
4. …especially machine learning and data science
5. Understand Python’s strengths (and weaknesses)

Your Questions

Your Questions

What is Python?

Your Questions

What is Python?

Will Python help me get a job / help me in the real world?

What is Python, anyway?

What is Python, anyway?

Guido Van Rossum
Former Benevolent Dictator for Life (BDFL)

What is Python, anyway?

Guido Van Rossum
Former Benevolent Dictator for Life (BDFL)

- Python 1: 1994
  - A middle ground between C and shell scripts.

What is Python, anyway?

Guido Van Rossum
Former Benevolent Dictator for Life (BDFL)

- Python 1: 1994
  - A middle ground between C and shell scripts.
- Python 2: 2000

What is Python, anyway?

Guido Van Rossum
Former Benevolent Dictator for Life (BDFL)

- Python 1: 1994
  - A middle ground between C and shell scripts.
- Python 2: 2000
- Python 3: 2008
  - We're using Python 3.8

Your Questions

Will Python help me get a job / help me in the real world?

Your Questions

Will Python help me get a job / help me in the real world?

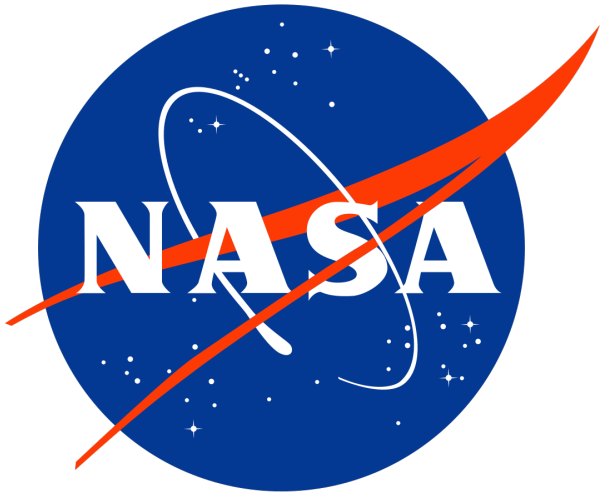
Python at Isitcom

-

Your Questions

Will Python help me get a job / help me in the real world?

Python in Government



Your Questions

Will Python help me get a job / help me in the real world?

Python in Business

Python is opinionated. This is its philosophy.

```
>>> import this
```

>>> import this

```
>>> import this

The Zen of Python, by Tim Peters
```

>>> import this

```
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
```

>>> import this

```
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
```

>>> import this

```
>>> 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

```
>>> 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

```
>>> 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

```
>>> 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

```
>>> 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

Case Study: Hello World

```
// Java
```

Case Study: Hello World

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

Case Study: Hello World

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

```
$ javac HelloWorld.java
$ java HelloWorld
Hello world!
```

Case Study: Hello World

```
// C++
```

Case Study: Hello World

```cpp
// C++
#include <iostream>
using namespace std;

int main() {
    cout << "Hello world!" << endl;
    return 0;
}
```

Case Study: Hello World

```cpp
// C++
#include <iostream>
using namespace std;

int main() {
    cout << "Hello world!" << endl;
    return 0;
}
```

```
$ g++ helloWorld.cpp
$ ./a.out
Hello world!
```

Case Study: Hello World

Case Study: Hello World

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

Case Study: Hello World

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

```
$ python helloworld.py
Hello world!
```

Course Logistics

Logistics

Schedule

Logistics

Schedule

Units

Logistics

Schedule

Units

github https://github.com/nevermind78

Logistics

Schedule

Units

Website https://github.com/nevermind78

Prereqs Algorithm

Logistics

Assignments 4 total (3 assignments + final project)

Logistics

Assignments	4 total (3 assignments + final project)
Grading	Checkmark scale, functionality & style You'll get credit if you average a check

Logistics

Assignments	4 total (3 assignments + final project)
Grading	Checkmark scale, functionality & style You'll get credit if you average a check
Late Days	Two 24-hour extensions

Logistics

Assignments	4 total (3 assignments + final project)
Grading	Checkmark scale, functionality & style You'll get credit if you average a check
Late Days	Two 24-hour extensions
Honor Code	Don't cheat

Logistics

Assignments	4 total (3 assignments + final project)
Grading	Checkmark scale, functionality & style You'll get credit if you average a check
Late Days	Two 24-hour extensions
Honor Code	Don't cheat

The Road Ahead…

The Road Ahead

Week 1: Welcome & Basics
Week 2: Data Structures
Week 3: Functions and Files
Week 4: Functional Programming
Week 5: Python & the Web

The Road Ahead

Week 6: NumPy
Week 7: Standard Library & Third Party Tools
Week 8: Plotting
Week 9: Machine Learning
Week 10: Advanced Topics

The Road Ahead

Week 6: NumPy
Week 7: Standard Library & Third Party Tools
Week 8: Plotting
Week 9: Machine Learning
Week 10: Advanced Topics

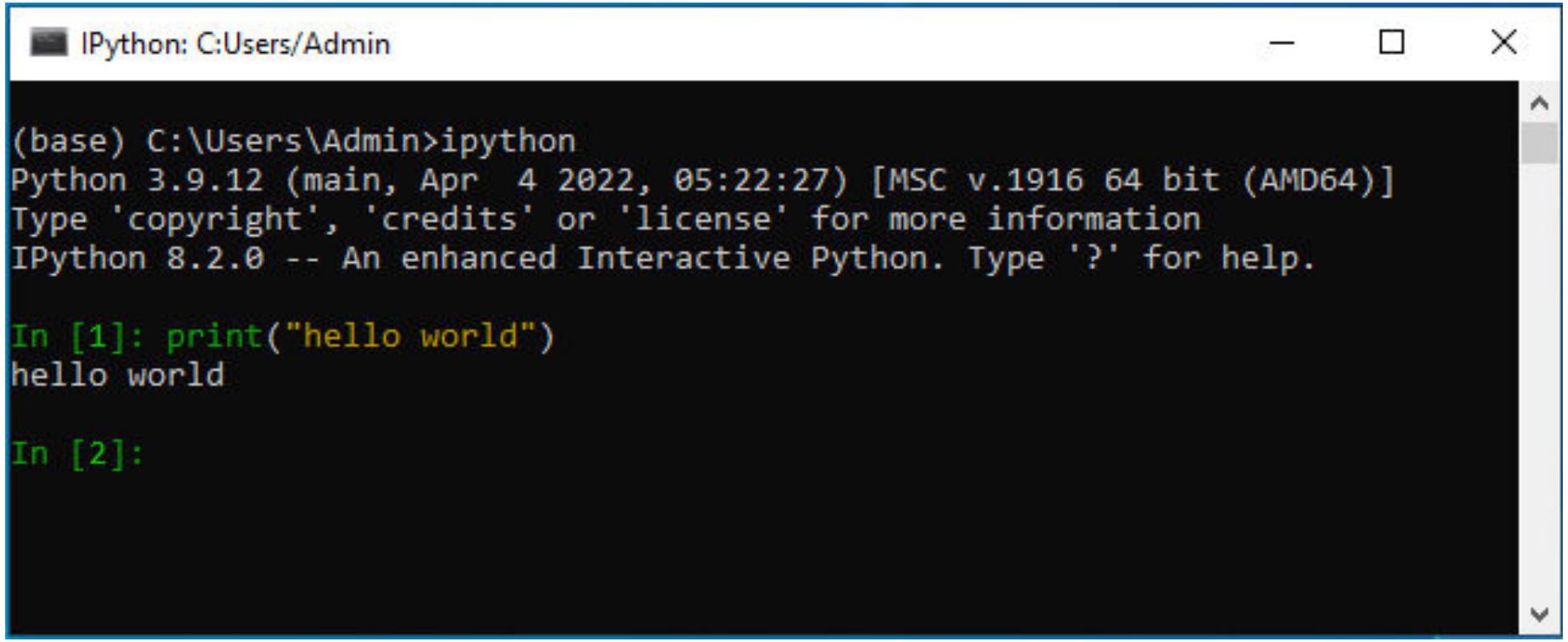
Weeks 8, 9, and 10 will be more focused on machine learning

Today's Content

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

Interactive Interpreter

Interactive Interpreter



It’s Up to Interpretation…

- Python is an *interpreted language*

It’s Up to Interpretation…

- Python is an *interpreted language*
  - Means no compiler, no executable file. The *Python Interpreter* parses the .py file and interprets the code as it goes.

It’s Up to Interpretation…

- Python is an *interpreted language*
  - Means no compiler. The *Python Interpreter* parses the .py file and interprets the code as it goes.
  - There is no prior compilation of source code into machine code.

It’s Up to Interpretation…

- Python is an *interpreted language*
  - Means no compiler. The *Python Interpreter* parses the .py file and interprets the code as it goes.
  - There is no prior compilation of source code into machine code.
- Features and *interactive interpreter* (what we just saw)

It’s Up to Interpretation…

- Python is an *interpreted language*
  - Means no compiler. The *Python Interpreter* parses the .py file and interprets the code as it goes.
  - There is no prior compilation of source code into machine code.
- Features and *interactive interpreter* (what we just saw)
  - Sandbox to experiment with Python.

It’s Up to Interpretation…

- Python is an *interpreted language*
  - Means no compiler. The *Python Interpreter* parses the .py file and interprets the code as it goes.
  - There is no prior compilation of source code into machine code.
- Features and *interactive interpreter* (what we just saw)
  - Sandbox to experiment with Python.
  - Shortens code-test-debug cycle.

It’s Up to Interpretation…

- Python is an *interpreted language*
  - Means no compiler. The *Python Interpreter* parses the .py file and interprets the code as it goes.
  - There is no prior compilation of source code into machine code.
- Features and *interactive interpreter* (what we just saw)
  - Sandbox to experiment with Python.
  - Shortens code-test-debug cycle.
  - This is the greatest thing in the world.

It's Up to Interpretation…

- Python is an *interpreted language*
  - Means no compiler. The *Python Interpreter* parses the .py file and interprets the code as it goes.
  - There is no prior compilation of source code into machine code.
- Features and *interactive interpreter* (what we just saw)
  - Sandbox to experiment with Python.
  - Shortens code-test-debug cycle.
  - This is the greatest thing in the world.
  - Assignment 0 will give you experience setting up – and using – the interactive interpreter!

Comments

Some Brief Commentary…

```
# A single-line comment in Python is denoted with the hash symbol.
```

Some Brief Commentary…

```
# A single-line comment in Python is denoted with the hash symbol.


"""
Multi-line comments
Lie between quotation marks
This is a haiku
"""
```

Variables and Types

Variables

```
int x = 5;       # C/C++/Java
```

Variables

```
int x = 5;       # C/C++/Java

x = 5            # Python - no type! No semicolon! What just happened?
```

A Type Dream

- Variables in Python are *dynamically typed.* They take on the type of the object they are representing.

A Type Dream

- Variables in Python are *dynamically typed.* They take on the type of the object they are representing.

```
>>> x = 1
>>>
```

A Type Dream

- Variables in Python are *dynamically typed.* They take on the type of the object they are representing.

```
>>> x = 1
>>> type(x)
```

A Type Dream

- Variables in Python are *dynamically typed.* They take on the type of the object they are representing.

```
>>> x = 1
>>> type(x)
<class 'int'>
>>>
```

A Type Dream

- Variables in Python are *dynamically typed.* They take on the type of the object they are representing.

```
>>> x = 1
>>> type(x)
<class 'int'>
>>> x = "Unicorn"
>>>
```

A Type Dream

- Variables in Python are *dynamically typed.* They take on the type of the object they are representing.

```
>>> x = 1
>>> type(x)
<class 'int'>
>>> x = "Unicorn"
>>> type(x)
```

A Type Dream

- Variables in Python are *dynamically typed.* They take on the type of the object they are representing.

```
>>> x = 1
>>> type(x)
<class 'int'>
>>> x = "Unicorn"
>>> type(x)
<class 'str'>
```

Numbers and Booleans

The Math of Least Resistance

- Python has only three numeric types: float, int, and complex.

```
5           # => 5    (int)
```

The Math of Least Resistance

- Python has only three numeric types: float, int, and complex.

```
5           # => 5   (int)
5.0         # => 5.0 (float)
```

The Math of Least Resistance

- Python has only three numeric types: float, int, and complex.

```
5           # => 5    (int)
5.0         # => 5.0  (float)

1 + 8       # => 9          (int)
```

The Math of Least Resistance

- Python has only three numeric types: float, int, and complex.

```
5           # => 5    (int)
5.0         # => 5.0  (float)

1 + 8       # => 9          (int)
8 + 5.1     # => 13.1       (float)
```

The Math of Least Resistance

- Python has only three numeric types: float, int, and complex.

```
5           # => 5    (int)
5.0         # => 5.0  (float)

1 + 8       # => 9          (int)
8 + 5.1     # => 13.1       (float)
2 * 4       # => 8          (int)
```

The Math of Least Resistance

- Python has only three numeric types: float, int, and complex.

```
5           # => 5    (int)
5.0         # => 5.0  (float)

1 + 8       # => 9          (int)
8 + 5.1     # => 13.1       (float)
2 * 4       # => 8          (int)
2 / 3       # => 0.666666   (float)
```

The Math of Least Resistance

- Python has only three numeric types: float, int, and complex.

```
5           # => 5    (int)
5.0         # => 5.0  (float)

1 + 8       # => 9          (int)
8 + 5.1     # => 13.1       (float)
2 * 4       # => 8          (int)
2 / 3       # => 0.666666   (float)

7 // 3      # => 2    (int; integer division)
```

The Math of Least Resistance

- Python has only three numeric types: float, int, and complex.

```
5           # => 5    (int)
5.0         # => 5.0  (float)

1 + 8       # => 9          (int)
8 + 5.1     # => 13.1       (float)
2 * 4       # => 8          (int)
2 / 3       # => 0.666666   (float)

7 // 3      # => 2    (int; integer division)
7  % 3      # => 1    (int; modulo operator)
```

The Math of Least Resistance

- Python has only three numeric types: float, int, and complex.

```
5           # => 5    (int)
5.0         # => 5.0  (float)

1 + 8       # => 9          (int)
8 + 5.1     # => 13.1       (float)
2 * 4       # => 8          (int)
2 / 3       # => 0.666666   (float)

7 // 3      # => 2    (int; integer division)
7  % 3      # => 1    (int; modulo operator)
3 ** 3      # => 27   (int; exponential operator)
```

Mathematical Assignment

- Appending an = to any of the prior operations allows for assignment based on the prior value of the variable.

```
x += 5          # => x = x + 5
x -= 3          # => x = x - 3
```

Mathematical Assignment

- Appending an = to any of the prior operations allows for assignment based on the prior value of the variable.

```
x += 5          # => x = x + 5
x -= 3          # => x = x - 3

x *= 2          # => x = x * 2
x /= 2          # => x = x / 2
```

Mathematical Assignment

- Appending an = to any of the prior operations allows for assignment based on the prior value of the variable.

```
x += 5          # => x = x + 5
x -= 3          # => x = x - 3

x *= 2          # => x = x * 2
x /= 2          # => x = x / 2

x //= 4         # => x = x // 4
x %= 3          # => x = x % 3
x **= 3         # => x = x ** 3
```

Veritas vos Libera*bit*

- The `bool` type is a subtype of `int`: `False == 0`, `True == 1`

```
True                    # => True
```

Veritas vos Libera*bit*

- The `bool` type is a subtype of `int`: `False == 0`, `True == 1`

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

Veritas vos Libera*bit*

- The `bool` type is a subtype of `int`: `False == 0`, `True == 1`

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

not True                # => False
```

Veritas vos Libera*bit*

- The `bool` type is a subtype of `int`: `False == 0`, `True == 1`

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

not True                # => False
True and False          # => False
```

Veritas vos Libera*bit*

- The `bool` type is a subtype of `int`: `False == 0`, `True == 1`

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

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

Veritas vos Libera*bit*

- The `bool` type is a subtype of `int`: `False == 0`, `True == 1`

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

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

5 == 5                  # => True
```

Veritas vos Libera*bit*

- The `bool` type is a subtype of `int`: `False == 0`, `True == 1`

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

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

5 == 5                  # => True
1 != 100                # => True
```

Veritas vos Libera*bit*

- The `bool` type is a subtype of `int`: `False == 0`, `True == 1`

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

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

5 == 5                  # => True
1 != 100                # => True
3 > 6                   # => False
```

Veritas vos Libera*bit*

- The `bool` type is a subtype of `int`: `False == 0`, `True == 1`

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

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

5 == 5                  # => True
1 != 100                # => True
3 > 6                   # => False
4 >= 9                  # => False
```

Veritas vos Libera*bit*

- The `bool` type is a subtype of `int`: `False == 0`, `True == 1`

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

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

5 == 5                  # => True
1 != 100                # => True
3 > 6                   # => False
4 >= 9                  # => False
3 > 2 > 1               # => True (3 > 2 and 2 > 1)
```

Strings and Lists

Strings

- Python doesn't have an explicit `char` type. Instead, `char`s are strings with length 1.
- Both `" ... "` and `' ... '` define string literals.
- Python strings are Unicode by default (yes – this means we can use emoji in Python strings! 😀 ❤️ 🚀 👍)

Strings

- Python doesn't have an explicit `char` type. Instead, `char`s are strings with length 1.
- Both " ... " and ' ... ' define string literals.
- Python strings are Unicode by default (yes – this means we can use emoji in Python strings! 😀 ❤️ 🚀 👍)

```
>>> x = "Parth loves 🦄"
>>>
```

Strings

- Python doesn't have an explicit `char` type. Instead, `char`s are strings with length 1.
- Both " ... " and ' ... ' define string literals.
- Python strings are Unicode by default (yes – this means we can use emoji in Python strings! 😀 ❤️ 🚀 👍)

```
>>> x = "Parth loves 🦄"
>>> x + "!"                    # String concatenation using "+"
```

Strings

- Python doesn't have an explicit `char` type. Instead, `char`s are strings with length 1.
- Both " ... " and ' ... ' define string literals.
- Python strings are Unicode by default (yes – this means we can use emoji in Python strings! 😀 ❤️ 🚀 👍)

```
>>> x = "Parth loves 🦄"
>>> x + "!"                    # String concatenation using "+"
"Parth loves 🦄!"
```

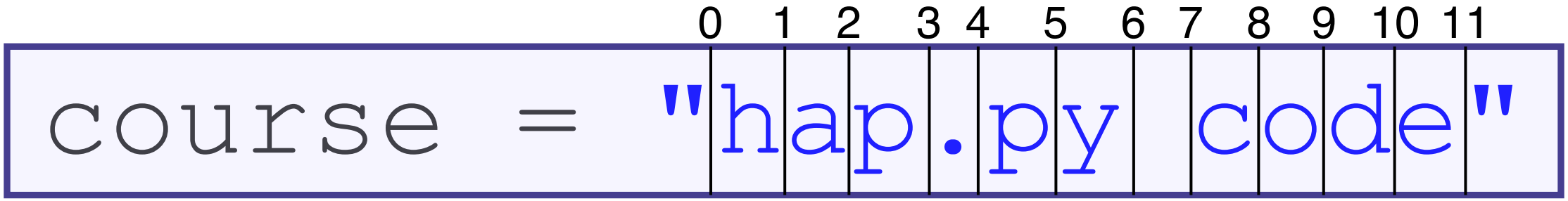
Strings

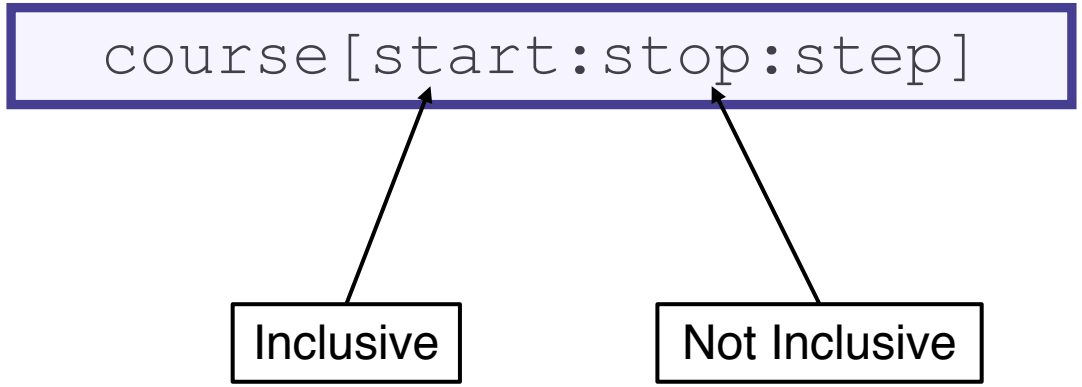
- Python doesn't have an explicit `char` type. Instead, `char`s are strings with length 1.
- Both " ... " and ' ... ' define string literals.
- Python strings are Unicode by default (yes – this means we can use emoji in Python strings! 😀 ❤️ 🚀 👍)

```
>>> x = "Parth loves 🦄"
>>> x + "!"                    # String concatenation using "+"
"Parth loves 🦄!"
```

- You can only concatenate strings with strings! If `x` is an integer, it must be cast as a string (`str(x)`) before being concatenated with another string.

Indexing Strings





Indexing Strings

```
>>> course = "hap.py code"
>>>
```

Indexing Strings

```
>>> course = "hap.py code"
>>> course[2]
```

Indexing Strings

```
>>> course = "hap.py code"
>>> course[2]                    # Index at individual elements
'p'
>>>
```

Indexing Strings

```
>>> course = "hap.py code"
>>> course[2]                    # Index at individual elements
'p'
>>> course[:3]
```

Indexing Strings

```
>>> course = "hap.py code"
>>> course[2]                    # Index at individual elements
'p'
>>> course[:3]                   # Index from beginning to element (exclusive)
'hap'
>>>
```

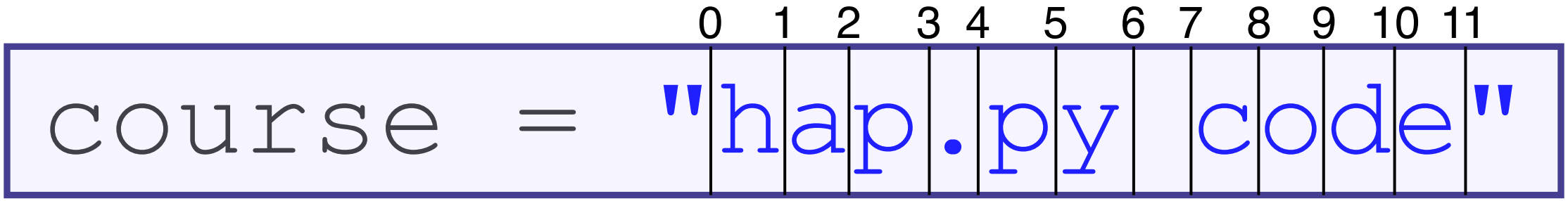
Indexing Strings

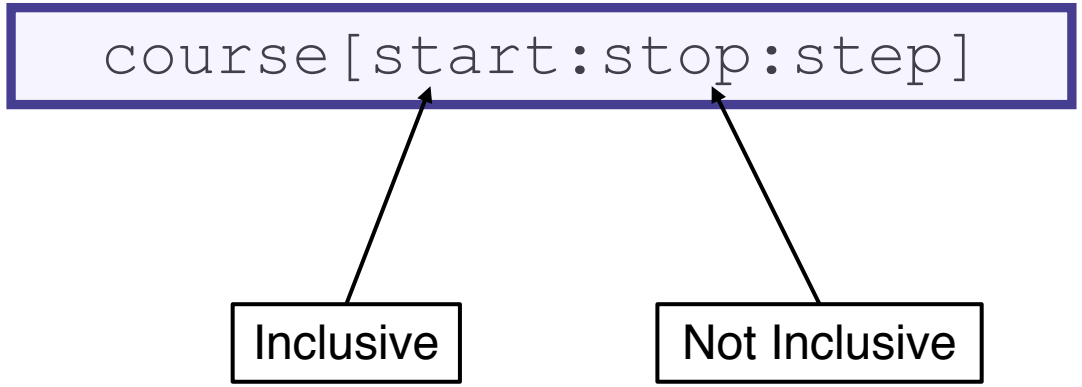
```
>>> course = "hap.py code"
>>> course[2]                    # Index at individual elements
'p'
>>> course[:3]                   # Index from beginning to element (exclusive)
'hap'
>>> course[5:]
```

Indexing Strings

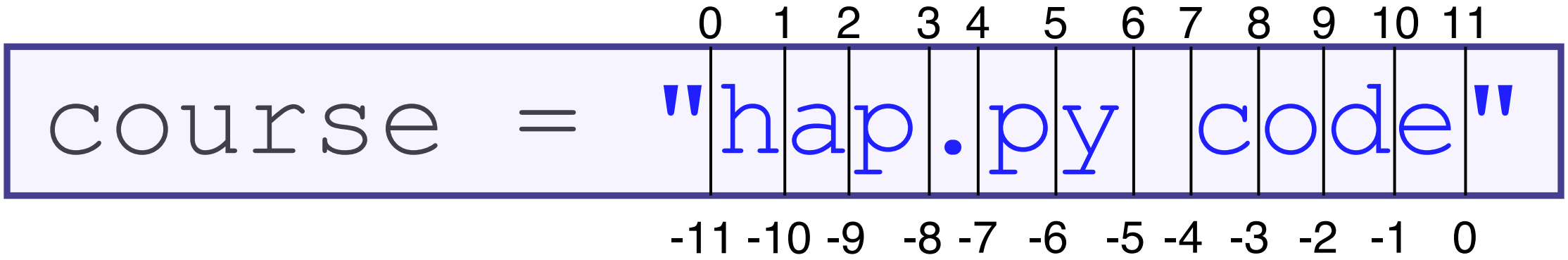
```
>>> course = "hap.py code"
>>> course[2]                    # Index at individual elements
'p'
>>> course[:3]                   # Index from beginning to element (exclusive)
'hap'
>>> course[5:]                   # Index from element to end (inclusive)
'y code'
```

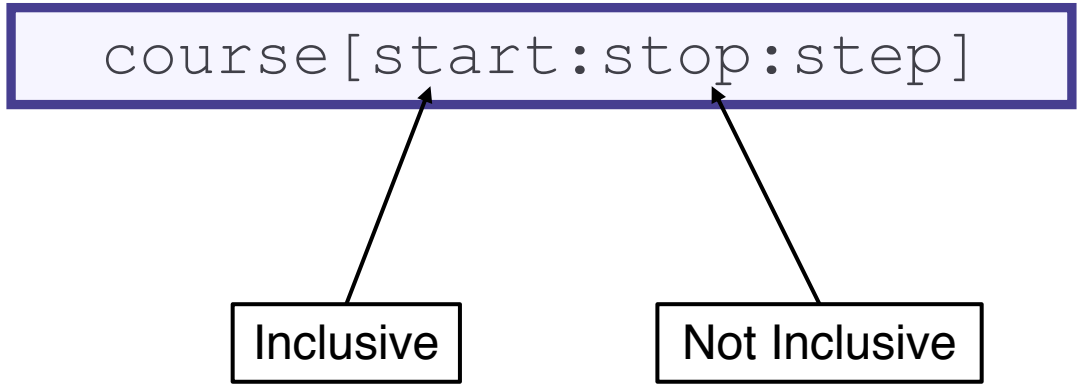
Indexing Strings





Indexing Strings





Indexing Strings

```
>>> course = "hap.py code"
>>> course[2]                    # Index at individual elements
'p'
>>> course[:3]                   # Index from beginning to element (exclusive)
'hap'
>>> course[5:]                   # Index from element to end (inclusive)
'y code'
>>> course[-2]
```

Indexing Strings

```
>>> course = "hap.py code"
>>> course[2]                    # Index at individual elements
'p'
>>> course[:3]                   # Index from beginning to element (exclusive)
'hap'
>>> course[5:]                   # Index from element to end (inclusive)
'y code'
>>> course[-2]                   # Negative indexing indexes back from the end
'd'
>>>
```

Indexing Strings

```
>>> course = "hap.py code"
>>> course[2]                    # Index at individual elements
'p'
>>> course[:3]                   # Index from beginning to element (exclusive)
'hap'
>>> course[5:]                   # Index from element to end (inclusive)
'y code'
>>> course[-2]                   # Negative indexing indexes back from the end
'd'
>>> course[1:8:2]
```

Indexing Strings

```
>>> course = "hap.py code"
>>> course[2]                    # Index at individual elements
'p'
>>> course[:3]                   # Index from beginning to element (exclusive)
'hap'
>>> course[5:]                   # Index from element to end (inclusive)
'y code'
>>> course[-2]                   # Negative indexing indexes back from the end
'd'
>>> course[1:8:2]                # Third parameter indicates a step size
'a.yc'
>>>
```

Indexing Strings

```
>>> course = "hap.py code"
>>> course[2]                    # Index at individual elements
'p'
>>> course[:3]                   # Index from beginning to element (exclusive)
'hap'
>>> course[5:]                   # Index from element to end (inclusive)
'y code'
>>> course[-2]                   # Negative indexing indexes back from the end
'd'
>>> course[1:8:2]                # Third parameter indicates a step size
'a.yc'
>>> course[8:1:-2]
```

Indexing Strings

```
>>> course = "hap.py code"
>>> course[2]                    # Index at individual elements
'p'
>>> course[:3]                   # Index from beginning to element (exclusive)
'hap'
>>> course[5:]                   # Index from element to end (inclusive)
'y code'
>>> course[-2]                   # Negative indexing indexes back from the end
'd'
>>> course[1:8:2]                # Third parameter indicates a step size
'a.yc'
>>> course[8:1:-2]               # Can do negative step sizes too
'o pp'
>>>
```

Indexing Strings

```
>>> course = "hap.py code"
>>> course[2]                    # Index at individual elements
'p'
>>> course[:3]                   # Index from beginning to element (exclusive)
'hap'
>>> course[5:]                   # Index from element to end (inclusive)
'y code'
>>> course[-2]                   # Negative indexing indexes back from the end
'd'
>>> course[1:8:2]                # Third parameter indicates a step size
'a.yc'
>>> course[8:1:-2]               # Can do negative step sizes too
'o pp'
>>> # Your turn! How would we reverse the string using this notation?
```

Indexing Strings

```
>>> course = "hap.py code"
>>> course[2]                    # Index at individual elements
'p'
>>> course[:3]                   # Index from beginning to element (exclusive)
'hap'
>>> course[5:]                   # Index from element to end (inclusive)
'y code'
>>> course[-2]                   # Negative indexing indexes back from the end
'd'
>>> course[1:8:2]                # Third parameter indicates a step size
'a.yc'
>>> course[8:1:-2]               # Can do negative step sizes too
'o pp'
>>> # Your turn! How would we reverse the string using this notation?
>>> course[::-1]
```

Lists – Python's ArrayList/Vector

```
# Create a new list
empty = []
letters = [“a”, “b”, “c”]
numbers = [1, 2, 3]
```

Lists – Python's ArrayList/Vector

```
# Create a new list
empty = []
letters = ["a", "b", "c"]
numbers = [1, 2, 3]

# Lists can contain elements of different types (even other lists!)
many_types = ["a", 2, 3, [4, 5]]

# Quick - what is many_types[3]?
```

Lists – Python's ArrayList/Vector

```
# Create a new list
empty = []
letters = ["a", "b", "c"]
numbers = [1, 2, 3]

# Lists can contain elements of different types (even other lists!)
many_types = ["a", 2, 3, [4, 5]]

# Quick - what is many_types[3]?
# [4, 5]
```

Lists – Python's ArrayList/Vector

```
# Create a new list
empty = []
letters = ["a", "b", "c"]
numbers = [1, 2, 3]

# Lists can contain elements of different types (even other lists!)
many_types = ["a", 2, 3, [4, 5]]

# Follow up - what is many_types[3][1]?
```

Lists – Python's ArrayList/Vector

```
# Create a new list
empty = []
letters = ["a", "b", "c"]
numbers = [1, 2, 3]

# Lists can contain elements of different types (even other lists!)
many_types = ["a", 2, 3, [4, 5]]

# Follow up - what is many_types[3][1]?
# 4
```

Lists – Python's ArrayList/Vector

```
# Create a new list
empty = []
letters = ["a", "b", "c"]
numbers = [1, 2, 3]

# Lists can contain elements of different types (even other lists!)
many_types = ["a", 2, 3, [4, 5]]

# Use the extend function to append elements of a list to another list
>>> numbers.extend([4, 5, 6])
>>> numbers
[1, 2, 3, 4]
```

Lists – Python's ArrayList/Vector

```
# Create a new list
empty = []
letters = ["a", "b", "c"]
numbers = [1, 2, 3]

# Lists can contain elements of different types (even other lists!)
many_types = ["a", 2, 3, [4, 5]]

# Use the extend function to append elements of a list to another list
>>> numbers.extend([4, 5, 6])
>>> numbers
[1, 2, 3, 4]
# Use the extend function to append elements of a list to another list
>>> numbers.extend([5, 6, 7])
>>> numbers
[1, 2, 3, 4, 5, 6, 7]
```

List Indexing and Slicing

```
# Create a new list
empty = []
letters = [“a”, “b”, “c”]
numbers = [1, 2, 3, 4]

# Indexing follows the exact same rules as string indexing
```

List Indexing and Slicing

```
# Create a new list
empty = []
letters = [“a”, “b”, “c”]
numbers = [1, 2, 3, 4]

# Indexing follows the exact same rules as string indexing
>>> letters[1]
‘b’
>>>
```

List Indexing and Slicing

```
# Create a new list
empty = []
letters = [“a”, “b”, “c”]
numbers = [1, 2, 3, 4]

# Indexing follows the exact same rules as string indexing
>>> letters[1]
‘b’
>>> numbers[1:-1]
```

List Indexing and Slicing

```
# Create a new list
empty = []
letters = [“a”, “b”, “c”]
numbers = [1, 2, 3, 4]

# Indexing follows the exact same rules as string indexing
>>> letters[1]
‘b’
>>> numbers[1:-1]
[2, 3]
>>>
```

List Indexing and Slicing

```
# Create a new list
empty = []
letters = [“a”, “b”, “c”]
numbers = [1, 2, 3, 4]

# Indexing follows the exact same rules as string indexing
>>> letters[1]
‘b’
>>> numbers[1:-1]
[2, 3]
>>> numbers[::2]
```

List Indexing and Slicing

```
# Create a new list
empty = []
letters = [“a”, “b”, “c”]
numbers = [1, 2, 3, 4]

# Indexing follows the exact same rules as string indexing
>>> letters[1]
‘b’
>>> numbers[1:-1]
[2, 3]
>>> numbers[::2]
[1, 3]
>>>
```

List Indexing and Slicing

```
# Create a new list
empty = []
letters = [“a”, “b”, “c”]
numbers = [1, 2, 3, 4]

# Indexing follows the exact same rules as string indexing
>>> letters[1]
‘b’
>>> numbers[1:-1]
[2, 3]
>>> numbers[::2]
[1, 3]
>>> numbers[1:3:-1]
```

List Indexing and Slicing

```
# Create a new list
empty = []
letters = [“a”, “b”, “c”]
numbers = [1, 2, 3, 4]

# Indexing follows the exact same rules as string indexing
>>> letters[1]
‘b’
>>> numbers[1:-1]
[2, 3]
>>> numbers[::2]
[1, 3]
>>> numbers[1:3:-1]
[]                                  # What just happened here?
>>>
```

Queries Over Collections

```
# Length
>>> len([])
```

Queries Over Collections

```
# Length
>>> len([])
0
```

Queries Over Collections

```
# Length
>>> len([])
0
>>> len("1LMX")
```

Queries Over Collections

```
# Length
>>> len([])
0
>>> len("1LMX")
4
```

Queries Over Collections

```
# Length
>>> len([])
0
>>> len("1LMX")
4
>>> len([1, 2, 3, "Parth"])
```

Queries Over Collections

```
# Length
>>> len([])
0
>>> len("1LMX")
4
>>> len([1, 2, 3, "Parth"])
4
```

Queries Over Collections

```
# Length
>>> len([])
0
>>> len("1LMX")
4
>>> len([1, 2, 3, "Parth"])
4


# Membership in a list
>>> 0 in [2, 3, 4]
```

Queries Over Collections

```
# Length
>>> len([])
0
>>> len("1LMX")
4
>>> len([1, 2, 3, "Parth"])
4


# Membership in a list
>>> 0 in [2, 3, 4]
False
```

Queries Over Collections

```
# Length
>>> len([])
0
>>> len("1LMX")
4
>>> len([1, 2, 3, "Parth"])
4


# Membership in a list
>>> 0 in [2, 3, 4]
False
>>> "5" in [3, 4, 5, "1LMX"]
```

Queries Over Collections

```
# Length
>>> len([])
0
>>> len("1LMX")
4
>>> len([1, 2, 3, "Parth"])
4


# Membership in a list
>>> 0 in [2, 3, 4]
False
>>> "5" in [3, 4, 5, "1LMX"]
False                                  # The integer 5 is a member, not the string!
```

Queries Over Collections

```
# Length
>>> len([])
0
>>> len("1LMX")
4
>>> len([1, 2, 3, "Parth"])
4


# Membership in a list
>>> 0 in [2, 3, 4]
False
>>> "5" in [3, 4, 5, "1LMX"]
False                                  # The integer 5 is a member, not the string!
# Checking for substrings using in keyword
>>> "tho" in "Python"
```

Queries Over Collections

```
# Length
>>> len([])
0
>>> len("1LMX")
4
>>> len([1, 2, 3, "Parth"])
4


# Membership in a list
>>> 0 in [2, 3, 4]
False
>>> "5" in [3, 4, 5, "1LMX"]
False                                  # The integer 5 is a member, not the string!
# Checking for substrings using in keyword
>>> "tho" in "Python"
True
```

Console I/O

Console I/O

```
# Read a string as user input and save it to a variable.

>>> fav_animal = input("What is your favourite animal? ")
```

Console I/O

```
# Read a string as user input and save it to a variable.

>>> fav_animal = input("What is your favourite animal? ")
What is your favourite animal?
```

Console I/O

```
# Read a string as user input and save it to a variable.

>>> fav_animal = input("What is your favourite animal? ")
What is your favourite animal? Unicorn
```

Console I/O

```
# Read a string as user input and save it to a variable.

>>> fav_animal = input("What is your favourite animal? ")
What is your favourite animal? Unicorn



>>> print("I also love " +  fav_animal + "s!")
```

Console I/O

```
# Read a string as user input and save it to a variable.

>>> fav_animal = input("What is your favourite animal? ")
What is your favourite animal? Unicorn



>>> print("I also love " +  fav_animal + "s!")
I also love Unicorns!
```

Control Flow

Control Flow

```
if the_world_is_friendly:
    print("Hello friendly world!")
```

- No parentheses necessary around the condition
- Colon at the end (no braces!)
- Use four spaces (or one tab) for indentation

`elif` and `else`

```
if the_world_is_friendly:
    print("Hello friendly world!")

elif the_world_is_happy:
    print("Hello happy world!")

else:
    print("The world is neither friendly nor happy! It just needs a hug.")
```

- Both `elif` and `else` are optional.
- Python has no `switch` statement, instead preferring `if/elif/else` chains.

Veritas vos Liberabit II

```
# All objects in Python are either “truthy” or “falsy”, meaning when cast
# as booleans, they evaluate to either True or False.
```

Veritas vos Liberabit II

```
# All objects in Python are either "truthy" or "falsy", meaning when cast
# as booleans, they evaluate to either True or False.

# Zero values and empty data structures are "falsy"
>>> bool(None)                          # => False
>>> bool(False)                         # => False
>>> bool(0)                             # => False
>>> bool(0.0)                           # => False
>>> bool("")                            # => False
>>> bool([])                            # => False
```

Veritas vos Liberabit II

```
# All objects in Python are either "truthy" or "falsy", meaning when cast
# as booleans, they evaluate to either True or False.

# Zero values and empty data structures are "falsy"
>>> bool(None)                          # => False
>>> bool(False)                         # => False
>>> bool(0)                             # => False
>>> bool(0.0)                           # => False
>>> bool("")                            # => False
>>> bool([])                            # => False

# Everything else is "truthy"!
>>> bool(42)                            # => True
>>> bool("Abdallah")                    # => True
>>> bool([1, 2, 3])                     # => True
```

Veritas vos Liberabit II

```
# All objects in Python are either "truthy" or "falsy", meaning when cast
# as booleans, they evaluate to either True or False.

# Zero values and empty data structures are "falsy"
>>> bool(None)                              # => False
>>> bool(False)                             # => False
>>> bool(0)                                 # => False
>>> bool(0.0)                               # => False
>>> bool("")                                # => False
>>> bool([])                                # => False

# Everything else is "truthy"!
>>> bool(42)                                # => True
>>> bool("Abdallah")                        # => True
>>> bool([1, 2, 3])                         # => True

# Quiz - is the following "truthy" or "falsy"?
>>> bool([False])
```

Veritas vos Liberabit II

```
# All objects in Python are either "truthy" or "falsy", meaning when cast
# as booleans, they evaluate to either True or False.

# Zero values and empty data structures are "falsy"
>>> bool(None)                              # => False
>>> bool(False)                             # => False
>>> bool(0)                                 # => False
>>> bool(0.0)                               # => False
>>> bool("")                                # => False
>>> bool([])                                # => False

# Everything else is "truthy"!
>>> bool(42)                                # => True
>>> bool("Abdallah")                        # => True
>>> bool([1, 2, 3])                         # => True

# Quiz - is the following "truthy" or "falsy"?
>>> bool([False])                           # => True
```

Checking for Emptiness

```
# Truthiness and falsiness can be useful in checking whether or not a
container is empty!

unicorns = []

if unicorns:
    frolic_with(unicorns)

else:
    print("No unicorns! 😥")
```

Loops

Loops

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

- No loop counter! Instead, the `item` variable takes on the value of sequential elements of the iterable.
- Iterables can be strings, lists, or one of any one of a number of other data types (which we will see more of on Wednesday!).

Looping Over Strings and Lists

```
# Loop over characters in a string
>>> for ch in "1LMX":
        print(ch)
```

Looping Over Strings and Lists

```
# Loop over characters in a string
>>> for ch in "1LMX":
        print(ch)
1
L
M
X
```

Looping Over Strings and Lists

```
# Loop over characters in a string
>>> for ch in "1LMX":
        print(ch)
1
L
M
X

>>> for num in [3, 1, 4, 1, 5]:
        print(num**2)
```

Looping Over Strings and Lists

```
# Loop over characters in a string
>>> for ch in "1LMX":
        print(ch)
1
L
M
X

>>> for num in [3, 1, 4, 1, 5]:
        print(num**2)
9
1
16
1
25
>>>
```

Range

- The `range` function generates an iterable over a range of numbers.
- Syntax: `range(stop)` or `range(start, stop[, step])`
  - Same rules as string slicing! `start` is inclusive, `stop` is exclusive.

```
>>> 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
```

Break and Continue

```
>>> for i in range(2, 10):
        if n == 6:
            break                   # Break breaks out of the smallest enclosing
        print(n)                    # for or while loop
```

Break and Continue

```
>>> for i in range(2, 10):
        if n == 6:
            break                   # Break breaks out of the smallest enclosing
        print(n)                    # for or while loop
2
3
4
5
```

Break and Continue

```
>>> for i in range(2, 10):
        if n == 6:
            break                   # Break breaks out of the smallest enclosing
        print(n)                    # for or while loop
2
3
4
5

for letter in "UNICORN":
    if letter in "UN":
        continue                    # Continue continues to the next iteration of
print(letter)                       # the loop
```

Break and Continue

```
>>> for i in range(2, 10):
        if n == 6:
            break                   # Break breaks out of the smallest enclosing
        print(n)                    # for or while loop
2
3
4
5

for letter in "UNICORN":
    if letter in "UN":
        continue                    # Continue continues to the next iteration of
print(letter)                       # the loop
I
C
O
R
```

Loops

- Syntax is very similar to the syntax for control flow.

```
while condition:
    do_stuff()
```

While Loops - Example

```
# Print powers of three below 10000
>>> n = 1


>>> while n < 10000:
        print(n)
        n *= 3
```

While Loops - Example

```
# Print powers of three below 10000
>>> n = 1


>>> while n < 10000:
        print(n)
        n *= 3

1
3
9
27
81
243
729
2187
6561
```

Functions

Functions

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

- The `def` keyword defines a function.
- Parameters do not have explicit types (unlike C, C++, Java).
- `return` is optional. If either return or its value are omitted, it implicitly returns `None`.

Example – Is A Number Prime?

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

Example – Is A Number Prime?

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

Example – Is A Number Prime?

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

Example – Is A Number Prime?

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

Example – Is A Number Prime?

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

Assignment Expressions

Assignment Expressions

- New feature in Python 3.8!
- Use `:=` to assign variables from an expression!

Assignment Expressions

- New feature in Python 3.8!
- Use `:=` to assign variables from an expression!

```
# Example: ordering food at a restaurant

# The old way of doing things...
order = input("What would you like to order?")
while order != "Nothing else":
    print("You're ordering: " + order)
    order = input("What would you like to order?")
```

Assignment Expressions

- New feature in Python 3.8!
- Use `:=` to assign variables from an expression!

```
# Example: ordering food at a restaurant

# The old way of doing things...
order = input("What would you like to order?")
while order != "Nothing else":
    print("You're ordering: " + order)
    order = input("What would you like to order?")

# Using assignment expressions!
while (order := input("What would you like to order?")) != "Nothing else":
    print("You're ordering: " + order)
```

Assignment Expressions

- New feature in Python 3.8!
- Use `:=` to assign variables from an expression!

```
# Example: ordering food at a restaurant

# The old way of doing things...
order = input("What would you like to order?")
while order != "Nothing else":
    print("You're ordering: " + order)
    order = input("What would you like to order?")

# Using assignment expressions!
while (order := input("What would you like to order?")) != "Nothing else":
    print("You're ordering: " + order)
```

- Remember – readability counts! Only use this if it doesn't compromise readability. 😊

Review

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