This is Section.

Week 6: Python

Gabe LeBlanc

Attendance Form: cs50.ly/section_attendance

Python	C
➔ high level	➔ low level
➔ easier to write	➔ more difficult to write
➔ interpreted	➔ compiled
➔ dynamically typed	➔ statically typed
➔ automatic memory management	➔ manual memory management

- Variables in Python do not require a type specifier, and do not need to be declared in advance.

- Variables in Python do not require a type specifier, and do not need to be declared in advance.

```
x = 54

phrase = "This is CS50"
```

- Variables in Python do not require a type specifier, and do not need to be declared in advance.

```
x = 54

phrase = 'This is CS50'
```

- Print statements are a lot similar.

```
name = “gabe”

print(name)
```

- Print statements are a lot similar.

```
name = “gabe”
print(f“Hello, {name}”)
```

- Conditionals look similar to C, but have slightly different keywords and are governed by indentation level, rather than curly braces.

- Conditionals look similar to C, but have slightly different keywords and are governed by indentation level, rather than curly braces.

```
if y < 43 or z == 15:
  # code goes here
```

- Conditionals look similar to C, but have slightly different keywords and are governed by indentation level, rather than curly braces.

```
if y < 43 or z == 15:
  # code goes here
```

- Conditionals look similar to C, but have slightly different keywords and are governed by indentation level, rather than curly braces.

```
if y < 43 or z == 15:
  # code goes here
```

- Conditionals look similar to C, but have slightly different keywords and are governed by indentation level, rather than curly braces.

```
if y < 43 or z == 15:
  # code goes here
```

- Conditionals look similar to C, but have slightly different keywords and are governed by indentation level, rather than curly braces.

```
if y < 43 or z == 15:
  # code goes here
elif y > 43 and x == 52:
  # more code goes here
```

- Conditionals look similar to C, but have slightly different keywords and are governed by indentation level, rather than curly braces.

```
if y < 43 or z == 15:
  # code goes here
elif y > 43 and x == 52:
  # more code goes here
```

- Conditionals look similar to C, but have slightly different keywords and are governed by indentation level, rather than curly braces.

```
if y < 43 or z == 15:
    # code goes here
elif y > 43 and x == 52:
    # more code goes here
```

- Conditionals look similar to C, but have slightly different keywords and are governed by indentation level, rather than curly braces.

```
if y < 43 or z == 15:
  # code goes here
else:
  # more code goes here
```

What about functions like get_int() or get_string()?

- To include files akin to what you did in C, use Python’s import!

What about functions like get_int() or get_string()?

- To include files akin to what you did in C, use Python’s import!

```
import cs50
```

What about functions like get_int() or get_string()?

- To include files akin to what you did in C, use Python's import!

```
import cs50
```

- Then you can use the functions inside of CS50's ***module***.

```
cs50.get_string()
```

What about functions like get_int() or get_string()?

- To include files akin to what you did in C, use Python’s import!

```
import cs50
```

- Then you can use the functions inside of CS50’s ***module***.

```
from cs50 import get_string
```

What about functions like get_int() or get_string()?

- To include files akin to what you did in C, use Python’s import!

```
import cs50
```

- Then you can use the functions inside of CS50’s ***module***.

- To run Python programs using your IDE’s interpreter, simply type:

What about functions like get_int() or get_string()?

- To include files akin to what you did in C, use Python’s import!

```
import cs50
```

- Then you can use the functions inside of CS50’s ***module***.

- To run Python programs using your IDE’s interpreter, simply type:

```
python <file>
```

What about functions like get_int() or get_string()?

- To include files akin to what you did in C, use Python’s import!

```
import cs50
```

- Then you can use the functions inside of CS50’s ***module***.

- To run Python programs using your IDE’s interpreter, simply type:

```
python mario.py
```

Hands On: code hello.py

(First part of the pset!)

- Loops primarily exist in two varieties only: `for` and `while`.

- Loops primarily exist in two varieties only: `for` and `while`.

```
counter = 0
while counter < 100:
   print(counter)
   counter += 1
```

- Loops primarily exist in two varieties only: `for` and `while`.

```
counter = 0
while counter < 100:
    print(counter)
    counter += 1
```

- Loops primarily exist in two varieties only: `for` and `while`.

```
counter = 0
while counter < 100:
    print(counter)
    counter += 1
```

- Loops primarily exist in two varieties only: `for` and `while`.

```
counter = 0
while counter < 100:
    print(counter)
    counter += 1
```

- Loops primarily exist in two varieties only: `for` and `while`.

```
for x in range(100):
print(counter)
```

- Loops primarily exist in two varieties only: `for` and `while`.

```
for x in range(100):
print(counter)
```

- Loops primarily exist in two varieties only: `for` and `while`.

```
for x in range(100):
   print(counter)
```

- Loops primarily exist in two varieties only: `for` and `while`.

```
for x in range(100):
   print(counter)
```

- Loops primarily exist in two varieties only: `for` and `while`.

```
for x in range(0, 100, 2):
    print(counter)
```

- Arrays in Python are more formally referred to as **lists**. And they are way more flexible than C’s. You can mix types, and they grow and shrink dynamically!

- Arrays in Python are more formally referred to as **lists**. And they are way more flexible than C’s. You can mix types, and they grow and shrink dynamically!

```
nums = []
```

- Arrays in Python are more formally referred to as **lists**. And they are way more flexible than C’s. You can mix types, and they grow and shrink dynamically!

```
nums = [1, 2, 3, 4]
```

- Arrays in Python are more formally referred to as **lists**. And they are way more flexible than C’s. You can mix types, and they grow and shrink dynamically!

```
items = [“apple”, “banana”]
for item in items:
    print(item)
```

- Arrays in Python are more formally referred to as **lists**. And they are way more flexible than C’s. You can mix types, and they grow and shrink dynamically!

```
nums = [x for x in range(100)]
```

- Arrays in Python are more formally referred to as **lists**. And they are way more flexible than C’s. You can mix types, and they grow and shrink dynamically!

```
nums = [x for x in range(100)]
```

- Arrays in Python are more formally referred to as **lists**. And they are way more flexible than C’s. You can mix types, and they grow and shrink dynamically!

```
nums = list()
```

- Arrays in Python are more formally referred to as **lists**. And they are way more flexible than C’s. You can mix types, and they grow and shrink dynamically!

```
nums = list()
```

- Arrays in Python are more formally referred to as **lists**. And they are way more flexible than C’s. You can mix types, and they grow and shrink dynamically!

```
nums = [1, 2, 3, 4]
```

- Arrays in Python are more formally referred to as **lists**. And they are way more flexible than C’s. You can mix types, and they grow and shrink dynamically!

```
nums = [1, 2, 3, 4]
nums.append(5)
```

- Arrays in Python are more formally referred to as **lists**. And they are way more flexible than C’s. You can mix types, and they grow and shrink dynamically!

```
nums = [1, 2, 3, 4]
nums.append(5)
```

- Arrays in Python are more formally referred to as **lists**. And they are way more flexible than C’s. You can mix types, and they grow and shrink dynamically!

```
nums = [1, True, "three", 4.0]
```

- Just like in C, strings are also just arrays! But, the tools you have to access things in them are considerably more powerful…

```
string = “helloooo”

print(string[0])
```

- Just like in C, strings are also just arrays! But, the tools you have to access things in them are considerably more powerful…

```
string = “helloooo”

print(string[0])

h
```

- Just like in C, strings are also just arrays! But, the tools you have to access things in them are considerably more powerful…

```
string = “helloooo”

print(string[-1])
```

- Just like in C, strings are also just arrays! But, the tools you have to access things in them are considerably more powerful…

```
string = “helloooo”

print(string[-1])

o
```

- Just like in C, strings are also just arrays! But, the tools you have to access things in them are considerably more powerful…

```
string = “helloooo”

print(string[4:7])
```

- Just like in C, strings are also just arrays! But, the tools you have to access things in them are considerably more powerful…

```
string = “hellоооо”

print(string[4:7])
```

```
ooo
```

- Just like in C, strings are also just arrays! But, the tools you have to access things in them are considerably more powerful…

```
string = “helloooo”

print(string[1:])
```

- Just like in C, strings are also just arrays! But, the tools you have to access things in them are considerably more powerful…

```
string = “helloooo”

print(string[1:])

elloooo
```

Hands On: code mario.c

Part 2 of the pset!

- ***Dictionaries*** in Python are akin to hash tables from C. They map key-value pairs. But the values don’t have to just be strings!

- ***Dictionaries*** in Python are akin to hash tables from C. They map key-value pairs. But the values don’t have to just be strings!

```
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}
```

- ***Dictionaries*** in Python are akin to hash tables from C. They map key-value pairs. But the values don’t have to just be strings!

```
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}
```

- ***Dictionaries*** in Python are akin to hash tables from C. They map key-value pairs. But the values don’t have to just be strings!

```
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}
```

- ***Dictionaries*** in Python are akin to hash tables from C. They map key-value pairs. But the values don’t have to just be strings!

```
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}
```

- ***Dictionaries*** in Python are akin to hash tables from C. They map key-value pairs. But the values don’t have to just be strings!

```
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}
```

- ***Dictionaries*** in Python are akin to hash tables from C. They map key-value pairs. But the values don’t have to just be strings!

```
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}
```

```
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}

pizzas["cheese"] = 8
```

```
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}

if pizzas["vegetables"] < 12:
    # do something
```

```
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}

pizzas["bacon"] = 14
```

- ***Dictionaries*** are not iterables on their own, but a list of a dictionary’s keys are iterable!

```
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}

for pie in pizzas:
    # use pie as a stand-in for your idea of "i" from C
```

```
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}

for pie in pizzas:
    print(pie)
```

```python
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}

for pie in pizzas:
    print(pie)
```

```
cheese
pepperoni
vegetable
buffalo chicken
```

```
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}

for pie, price in pizzas.items():
    print(price)
```

```
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}

for pie, price in pizzas.items():
    print(price)
```

```
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}

for pie, price in pizzas.items():
    print(price)
```

```
9
10
11
12
```

```python
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}

for pie, price in pizzas.items():
    print(f"A whole {pie} pizza costs ${price}.")
```

```python
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}

for pie, price in pizzas.items():
    print(f"A whole {pie} pizza costs ${price}.")
```

```
A whole cheese pizza costs $9.
A whole pepperoni pizza costs $10.
A whole vegetable pizza costs $11.
A whole buffalo chicken pizza costs $12.
```

- Functions behave nearly identically to C, and just have a different syntax.

- Functions behave nearly identically to C, and just have a different syntax.

```
def square(x):
    return x * x
```

- Functions behave nearly identically to C, and just have a different syntax.

```
def square(x):
    return x ** 2
```

- Functions behave nearly identically to C, and just have a different syntax.

```
def square(x):
   return x ** 2
```

- Functions behave nearly identically to C, and just have a different syntax.

```
def division(x, y):
    return x / y
```

- Functions behave nearly identically to C, and just have a different syntax.

```
def floor_division(x, y):
    return x // y
```

- Functions behave nearly identically to C, and just have a different syntax.

```
def square(x):
   return x ** 2

# note indentation, no longer in function
print(square(5))
```

- Functions behave nearly identically to C, and just have a different syntax.

```
def square(x):
    return x ** 2

# note indentation, no longer in function
print(square(5))
```

- Python is ***object-oriented***.

- Think of an object like a C structure. They contain a number of fields which we’ll now start calling *properties*, but they also contain **functions** that might apply only to those objects. We call those *methods*.

- You’ve seen us use several methods already!

```
pizzas = {
    "cheese": 9,
    "pepperoni": 10,
    "vegetable": 11,
    "buffalo chicken": 12
}

for pie, price in pizzas.items():
    print(f"A whole {pie} pizza costs ${price}.")
```

```
nums = [1, 2, 3, 4]
nums.append(5)
```

- Lists (and indeed most native things in Python) are already objects, though it is also possible to define your own objects.

- To create a new type of object you define a Python ***class***. The only method required of a class is the method one uses to create an object of that type, which we normally call a ***constructor***.

```
class Student():

    # constructor, and this is two underscores on each side
    def __init__(self, name, id):
        self.name = name
        self.id = id

    # method to change a student’s ID
    def changeID(self, id):
        self.id = id

    # method to print the object. No parameters but still need self
    def print(self):
        print(f"{self.name} has ID {self.id}")
```

```python
class Student():

    # constructor, and this is two underscores on each side
    def __init__(self, name, id):
        self.name = name
        self.id = id

    # method to change a student’s ID
    def changeID(self, id):
        self.id = id

    # method to print the object. No parameters but still need self
    def print(self):
        print(f"{self.name} has ID {self.id}")
```

```
class Student():

    # constructor, and this is two underscores on each side
    def __init__(self, name, id):
        self.name = name
        self.id = id

    # method to change a student’s ID
    def changeID(self, id):
        self.id = id

    # method to print the object. No parameters but still need self
    def print(self):
        print(f"{self.name} has ID {self.id}")

jane = Student("Jane", 10)
jane.print()
jane.changeID(11)
jane.print()
```

That’s a lot of syntax. Gabe doesn’t remember it all, and you don’t need to either. **Google is your friend.**