Python Syntax

Python Syntax

- Python is an example of a very commonly-used modern programming language.
  - C was first released in 1972, Python in 1991.
- Python is an excellent and versatile language choice for making complex C operations much simpler.
  - String manipulation
  - Networking
- Fortunately, Python is heavily inspired by C (its primary interpreter, *Cpython*, is actually written in C) and so the syntax should be a shallow learning curve.

Python Syntax

- To start writing Python, open up a file with the .py file extension.

- Unlike a C program, which typically has to be compiled before you can run it, a Python program can be run without explicitly compiling it first.

- Important note: In CS50, we teach **Python 3**. (Not Python 2, which is also still fairly popular.)

Python Syntax

- **Variables**

- Python variables have two big differences from C.
  - No type specifier.
  - Declared by initialization only.

Python Syntax

- **Variables**

- Python variables have two big differences from C.
  - No type specifier.
  - Declared by initialization only.

```
int x = 54;
```

Python Syntax

- **Variables**

- Python variables have two big differences from C.
  - No type specifier.
  - Declared by initialization only.

~~`int x = 54;`~~

Python Syntax

- **Variables**

- Python variables have two big differences from C.
  - No type specifier.
  - Declared by initialization only.

```
x = 54
```

Python Syntax

- **Variables**

- Python variables have two big differences from C.
  - No type specifier.
  - Declared by initialization only.
  - Python statements needn't end with semicolons!

```
x = 54
```

Python Syntax

- **Variables**

- Python variables have two big differences from C.
  - No type specifier.
  - Declared by initialization only.

```
string phrase = "This is CS50";
```

Python Syntax

- **Variables**

- Python variables have two big differences from C.
  - No type specifier.
  - Declared by initialization only.

~~`string phrase = "This is CS50";`~~

Python Syntax

- **Variables**

- Python variables have two big differences from C.
  - No type specifier.
  - Declared by initialization only.

```
phrase = "This is CS50"
```

Python Syntax

- **Variables**

- Python variables have two big differences from C.
  - No type specifier.
  - Declared by initialization only.

```
phrase = 'This is CS50'
```

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use, but they look a little bit different now.

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use, but they look a little bit different now.

```
if (y < 43 || z == 15)
{
    // code goes here
}
```

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use, but they look a little bit different now.

```
if (y < 43 || z == 15)
{
    // code goes here
}
```

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use, but they look a little bit different now.

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

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use, but they look a little bit different now.

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

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use, but they look a little bit different now.

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

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use.

```
if (y < 43 && z == 15)
{
     // code block 1
}
else
{
     // code block 2
}
```

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use.

```
if (y < 43 && z == 15)
{
    // code block 1
}
else
{
    // code block 2
}
```

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use.

```
if y < 43 and z == 15:
      # code block 1
else:
      # code block 2
```

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use.

```
if y < 43 and z == 15:
      # code block 1
else:
      # code block 2
```

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use.

```
if (coursenum == 50)
{
      // code block 1
}
else if (coursenum != 51)
{
      // code block 2
}
```

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use.

```
if (coursenum == 50)
{
    // code block 1
}
else if (coursenum != 51)
{
    // code block 2
}
```

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use.

```
if coursenum == 50:
      # code block 1
elif not coursenum == 51:
      # code block 2
```

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use.

```
if coursenum == 50:
      # code block 1
elif not coursenum == 51:
      # code block 2
```

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use.

```
if coursenum == 50:
      # code block 1
elif not coursenum == 51:
      # code block 2
```

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use.

```
char var = get_char();
bool alphabetic = isalpha(var) ? true : false;
```

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use.

```
char var = get_char();
bool alphabetic = isalpha(var) ? true : false;
```

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use.

```
letters_only = True if input().isalpha() else False
```

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use.

```
letters_only = True if input().isalpha() else False
```

Python Syntax

- **Conditionals**

- All of the old favorites from C are still available for you to use.

```
letters_only = True if input().isalpha() else False
```

Python Syntax

- **Loops**

- Two varieties: `while` and `for`

Python Syntax

- **Loops**

- Two varieties: `while` and `for`

```
int counter = 0;
while (counter < 100)
{
    printf("%i\n", counter);
    counter++;
}
```

Python Syntax

- **Loops**

- Two varieties: `while` and `for`

```
int counter = 0;
while (counter < 100)
{
    printf("%i\n", counter);
    counter++;
}
```

Python Syntax

- **Loops**

- Two varieties: `while` and `for`

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

Python Syntax

- **Loops**

- Two varieties: `while` and `for`

```
for (int x = 0; x < 100; x++)
{
    printf("%i\n", x);
}
```

Python Syntax

- **Loops**

- Two varieties: `while` and `for`

```
for (int x = 0; x < 100; x++)
{
    printf("%i\n", x);
}
```

Python Syntax

- **Loops**

- Two varieties: `while` and `for`

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

Python Syntax

- **Loops**

- Two varieties: `while` and `for`

```
for (int x = 0; x < 100; x += 2)
{
    printf("%i\n", x);
}
```

Python Syntax

- **Loops**

- Two varieties: `while` and `for`

```
for (int x = 0; x < 100; x += 2)
{
    printf("%i\n", x);
}
```

Python Syntax

- **Loops**

- Two varieties: `while` and `for`

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

Python Syntax

- **Arrays**

- Here's where things really start to get a lot better than C.

- Python arrays (more appropriately known as ***lists***) are **not** fixed in size; they can grow or shrink as needed, and you can always tack extra elements onto your array and splice things in and out easily.

Python Syntax

- ~~**Arrays**~~ **Lists**

- Here's where things really start to get a lot better than C.

- Python arrays (more appropriately known as ***lists***) are **not** fixed in size; they can grow or shrink as needed, and you can always tack extra elements onto your array and splice things in and out easily.

Python Syntax

- **Lists**

- Declaring a list is pretty straightforward.

```
nums = []
```

Python Syntax

- **Lists**

- Declaring a list is pretty straightforward.

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

Python Syntax

- **Lists**

- Declaring a list is pretty straightforward.

```
nums = [x for x in range(500)]
```

Python Syntax

- **Lists**

- Declaring a list is pretty straightforward.

```
nums = list()
```

Python Syntax

- **Lists**

- Tacking on to an existing list can be done a few ways:

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

Python Syntax

- **Lists**

- Tacking on to an existing list can be done a few ways:

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

Python Syntax

- **Lists**

- Tacking on to an existing list can be done a few ways:

```
nums = [1, 2, 3, 4]
nums[len(nums):] = [5]
```

Python Syntax

- **Lists**

- Tacking on to an existing list can be done a few ways:

```
nums = [1, 2, 3, 4]
nums[len(nums):] = [5]
```

Python Syntax

- **Tuples**

- Python also has a data type that is not quite like anything comparable to C, a *tuple*.

- Tuples are ordered, immutable sets of data; they are great for associating collections of data, sort of like a struct in C, but where those values are unlikely to change.

Python Syntax

- **Tuples**

- Here is a list of tuples:

Python Syntax

- **Tuples**

- Here is a list of tuples:

```
presidents = [
    ("George Washington", 1789),
    ("John Adams", 1797),
    ("Thomas Jefferson", 1801),
    ("James Madison", 1809)
]
```

Python Syntax

- **Tuples**

- This list is iterable as well:

```
presidents = [
    ("George Washington", 1789),
    ("John Adams", 1797),
    ("Thomas Jefferson", 1801),
    ("James Madison", 1809)
]
```

Python Syntax

- **Tuples**

```
presidents = [
    ("George Washington", 1789),
    ("John Adams", 1797),
    ("Thomas Jefferson", 1801),
    ("James Madison", 1809)
]
```

- This list is iterable as well:

```
for prez, year in presidents:
    print("In {1}, {0} took office".format(prez, year))
```

Python Syntax

```
presidents = [
    ("George Washington", 1789),
    ("John Adams", 1797),
    ("Thomas Jefferson", 1801),
    ("James Madison", 1809)
]
```

- **Tuples**

- This list is iterable as well:

```
for prez, year in presidents:
    print("In {1}, {0} took office".format(prez, year))
```

Python Syntax

```
presidents = [
    ("George Washington", 1789),
    ("John Adams", 1797),
    ("Thomas Jefferson", 1801),
    ("James Madison", 1809)
]
```

- **Tuples**

- This list is iterable as well:

```
for prez, year in presidents:
    print("In {1}, {0} took office".format(prez, year))
```

```
In 1789, George Washington took office
In 1797, John Adams took office
In 1801, Thomas Jefferson took office
In 1809, James Madison took office
```

Python Syntax

- **Dictionaries**

- Python also has built in support for **dictionaries**, allowing you to specify list indices with words or phrases (*keys*), instead of integers, which you were restricted to in C.

Python Syntax

- **Dictionaries**

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

Python Syntax

- **Dictionaries**

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

Python Syntax

- **Dictionaries**

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

Python Syntax

- **Dictionaries**

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

Python Syntax

- **Dictionaries**

```
pizzas["cheese"] = 8
```

Python Syntax

- **Dictionaries**

```
pizzas["cheese"] = 8

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

Python Syntax

- **Dictionaries**

```
pizzas["cheese"] = 8

if pizza["vegetables"] < 12:
    # do something

pizzas["bacon"] = 14
```

Python Syntax

- Python also has built in support for **dictionaries**, allowing you to specify list indices with words or phrases (*keys*), instead of integers, which you were restricted to in C.

- But this creates a somewhat new problem… how do we iterate through a dictionary? We don't have indexes ranging from [0, *n*-1] anymore.

Python Syntax

- **Loops (redux)**

- The for loop in Python is extremely flexible!

```
for pie in pizzas:
    # use pie in here as a stand-in for "i"
```

Python Syntax

- **Loops (redux)**

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

Python Syntax

- **Loops (redux)**

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

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

Python Syntax

- **Loops (redux)**

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

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

```
cheese
vegetable
buffalo chicken
pepperoni
```

Python Syntax

- **Loops (redux)**

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

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

Python Syntax

- **Loops (redux)**

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

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

Python Syntax

- **Loops (redux)**

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

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

```
12
10
9
11
```

Python Syntax

- **Loops (redux)**

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

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

Python Syntax

- **Loops (redux)**

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

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

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

Python Syntax

- **Printing and variable interpolation**

- `format` gives one way to interpolate variables into our printed statements in a very `printf`-like way, but there are others.

```
print("A whole {} pizza costs ${}".format(pie, price))
```

Python Syntax

- **Printing and variable interpolation**

- `format` gives one way to interpolate variables into our printed statements in a very `printf`-like way, but there are others.

```
print("A whole {} pizza costs ${}".format(pie, price))

print("A whole " + pie + " pizza costs $" + str(price))
```

Python Syntax

- **Printing and variable interpolation**

- `format` gives one way to interpolate variables into our printed statements in a very `printf`-like way, but there are others.

```
print("A whole {} pizza costs ${}".format(pie, price))

print("A whole " + pie + " pizza costs $" + str(price))
```

Python Syntax

- **Printing and variable interpolation**

- `format` gives one way to interpolate variables into our printed statements in a very `printf`-like way, but there are other/s.

```
print("A whole {} pizza costs ${}".format(pie, price))

print("A whole " + pie + " pizza costs $" + str(price))

# you may see this, but avoid; deprecated
print("A whole %s pizza costs $%2d" % (pie, price))
```

Python Syntax

- **Functions**

- Python has support for functions as well. Like variables, we don't need to specify the return type of the function (because it doesn't matter), nor the data types of any parameters (ditto).

- All functions are introduced with the `def` keyword.
  - Also, no need for `main`; the interpreter reads from top to bottom!
  - If you wish to define `main` nonetheless (and you might want to!), you must at the very end of your code have:

```
if __name__ == "__main__":
    main()
```

Python Syntax

- **Functions**

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

Python Syntax

- **Functions**

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

Python Syntax

- **Functions**

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

Python Syntax

- **Functions**

```
def square(x):
    result = 0
    for i in range(0, x):
        result += x
    return result
```

Python Syntax

- **Functions**

```
def square(x):
    result = 0
    for i in range(0, x):
        result += x
    return result

print(square(5))
```

Python Syntax

- **Objects**

- Python is an *object-oriented* programming language.

- An object is sort of analogous to a C structure.

Python Syntax

- **Objects**

- C structures contain a number of *fields*, which we might also call *properties*.
  - But the properties themselves can not ever stand on their own.

Python Syntax

```
struct car
{
    int year;
    char *model;
}
```

- **Objects**

- C structures contain a number of *fields*, which we might also call *properties*.
  - But the properties themselves can not ever stand on their own.

Python Syntax

```
struct car
{
    int year;
    char *model;
}
```

- **Objects**

- C structures contain a number of *fields*, which we might also call *properties*.
  - But the properties themselves can not ever stand on their own.

```
struct car herbie;
```

Python Syntax

```
struct car
{
    int year;
    char *model;
}
```

- **Objects**

- C structures contain a number of *fields*, which we might also call *properties*.
  - But the properties themselves can not ever stand on their own.

```
struct car herbie;
herbie.year = 1963;
herbie.model = "Beetle";
```

Python Syntax

```
struct car
{
    int year;
    char *model;
}
```

- **Objects**

- C structures contain a number of *fields*, which we might also call *properties*.
  - But the properties themselves can not ever stand on their own.

```
struct car herbie;
year = 1963;
model = "Beetle";
```

Python Syntax

```
struct car
{
    int year;
    char *model;
}
```

- **Objects**

- C structures contain a number of *fields*, which we might also call *properties*.
  - But the properties themselves can not ever stand on their own.

```
struct car herbie;
year = 1963;
model = "Beetle";
```

Python Syntax

- **Objects**

- C structures contain a number of *fields*, which we might also call *properties*.
  - But the properties themselves can not ever stand on their own.

- Objects, meanwhile, have properties but also *methods*, or functions that are inherent to the object, and mean nothing outside of it. You define the methods inside the object also.
  - Thus, properties and methods don't ever stand on their own.

Python Syntax

- **Objects**

```
function(object);
```

Python Syntax

- **Objects**

~~`function(object);`~~

Python Syntax

- **Objects**

```
object.method()
```

Python Syntax

- **Objects**

- You define a type of object using the `class` keyword in Python.
- Classes require an initialization function, also more-generally known as a *constructor*, which sets the starting values of the properties of the object.
- In defining each method of an object, `self` should be its first parameter, which stipulates on what object the method is called.

Python Syntax

- **Objects**

```
class Student():

    def __init__(self, name, id):
        self.name = name
        self.id = id

    def changeID(self, id):
        self.id = id

    def print(self):
        print("{} - {}".format(self.name, self.id))
```

Python Syntax

- **Objects**

```
class Student():

    def __init__(self, name, id):
        self.name = name
        self.id = id

    def changeID(self, id):
        self.id = id

    def print(self):
        print("{} - {}".format(self.name, self.id))
```

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

Python Syntax

- **Style**

- If you haven't noticed, good style is **crucial** in Python.
- Tabs and indentation actually matter in this language, and things will not work the way you intend for them to if you disregard styling!
- Good news? No more curly braces to delineate blocks!
  - Now they just are used to declare dictionaries.

Python Syntax

- **Including files**

- Just like C programs can consist of multiple files to form a single program, so can Python programs tie files together.

```
#include <cs50.h>
```

Python Syntax

- **Including files**

- Just like C programs can consist of multiple files to form a single program, so can Python programs tie files together.

~~`#include <cs50.h>`~~

Python Syntax

- **Including files**

- Just like C programs can consist of multiple files to form a single program, so can Python programs tie files together.

```
import cs50
```

Python Syntax

- **Including files**

- Just like C programs can consist of multiple files to form a single program, so can Python programs tie files together.

```
cs50.get_int()
cs50.get_float()
cs50.get_string()
```

Python Syntax

- Python programs can be prewritten in .py files, but you can also write and test short Python snippets using the Python interpreter from the command line.

- All that is required is that the Python interpreter is installed on the system you wish to run your Python programs on.

Python Syntax

- To run your Python program through the Python interpreter at the command-line, simply type

```
python <file>
```

- and your program will run through the interpreter, which will execute everything inside of the file, top to bottom.

Python Syntax

- You can also make your programs look a lot more like C programs when they execute by adding a **shebang** to the top of your Python files, which automatically finds and executes the interpreter for you.

```
#!/usr/bin/env python3
```

- If you do this, you need to change the **permissions** on your file as well using the Linux command `chmod` as follows:

```
chmod a+x <file>
```