CS 115 Lecture 21

Classes, data structures, and C++

Neil Moore

Department of Computer Science
University of Kentucky
Lexington, Kentucky 40506
`neil@cs.uky.edu`

3 December 2015

Files versus lists

Differences between files and lists:

Files	Lists
Are permanent: persist beyond the end of your program.	Are temporary: only last until the program ends.

Files versus lists

Differences between files and lists:

Files	Lists
Are permanent: persist beyond the end of your program.	Are temporary: only last until the program ends.
Can be bigger than memory.	Must fit in memory.

Files versus lists

Differences between files and lists:

Files	Lists
Are permanent: persist beyond the end of your program.	Are temporary: only last until the program ends.
Can be bigger than memory.	Must fit in memory.
Are slow: access times from microseconds to milliseconds.	Are fast: access times from nanoseconds to microseconds.

Files versus lists

Differences between files and lists:

Files	Lists
Are permanent: persist beyond the end of your program.	Are temporary: only last until the program ends.
Can be bigger than memory.	Must fit in memory.
Are slow: access times from microseconds to milliseconds.	Are fast: access times from nanoseconds to microseconds.
Work best with sequential access.	Support fast random access.

Files versus lists

Differences between files and lists:

Files	Lists
Are permanent: persist beyond the end of your program.	Are temporary: only last until the program ends.
Can be bigger than memory.	Must fit in memory.
Are slow: access times from microseconds to milliseconds.	Are fast: access times from nanoseconds to microseconds.
Work best with sequential access.	Support fast random access.
Contain either bytes (binary files) or characters arranged in lines (text files).	Can contain any type.

Files versus lists

Differences between files and lists:

Files	Lists
Are permanent: persist beyond the end of your program.	Are temporary: only last until the program ends.
Can be bigger than memory.	Must fit in memory.
Are slow: access times from microseconds to milliseconds.	Are fast: access times from nanoseconds to microseconds.
Work best with sequential access.	Support fast random access.
Contain either bytes (binary files) or characters arranged in lines (text files).	Can contain any type.

Defining data structures

Sometimes it is helpful to group together several pieces of data.

Defining data structures

Sometimes it is helpful to group together several pieces of data.

- For example, employee name, email, and salary.

Defining data structures

Sometimes it is helpful to group together several pieces of data.

- For example, employee name, email, and salary.
- Song: title, artist, album, track number, . . .

Defining data structures

Sometimes it is helpful to group together several pieces of data.

- For example, employee name, email, and salary.
- Song: title, artist, album, track number, ...
- We can use one variable for each piece of information.
  - Or a list of records if we need multiple employees, songs, etc.

Defining data structures

Sometimes it is helpful to group together several pieces of data.

- For example, employee name, email, and salary.
- Song: title, artist, album, track number, . . .
- We can use one variable for each piece of information.
  - Or a list of records if we need multiple employees, songs, etc.
  - But it is much more convenient to group them together.
- In Python, we can group together these data into **objects**.

Defining data structures

Sometimes it is helpful to group together several pieces of data.

- For example, employee name, email, and salary.
- Song: title, artist, album, track number, . . .
- We can use one variable for each piece of information.
  - Or a list of records if we need multiple employees, songs, etc.
  - But it is much more convenient to group them together.
- In Python, we can group together these data into **objects**.
- An object has two components:
  - **Data** ("attributes", "members", "instance variables", "fields")

Defining data structures

Sometimes it is helpful to group together several pieces of data.

- For example, employee name, email, and salary.
- Song: title, artist, album, track number, . . .
- We can use one variable for each piece of information.
  - Or a list of records if we need multiple employees, songs, etc.
  - But it is much more convenient to group them together.
- In Python, we can group together these data into **objects**.
- An object has two components:
  - **Data** ("attributes", "members", "instance variables", "fields")
  - **Operations** that can be performed on the object ("methods", "member functions")

Defining data structures

Sometimes it is helpful to group together several pieces of data.

- For example, employee name, email, and salary.
- Song: title, artist, album, track number, ...
- We can use one variable for each piece of information.
  - Or a list of records if we need multiple employees, songs, etc.
  - But it is much more convenient to group them together.
- In Python, we can group together these data into **objects**.
- An object has two components:
  - **Data** ("attributes", "members", "instance variables", "fields")
  - **Operations** that can be performed on the object ("methods", "member functions")
- We've already seen several kinds of objects:
  - Graphics shapes.
  - Strings.
  - Lists.
  - File objects.

Defining data structures

Sometimes it is helpful to group together several pieces of data.

- For example, employee name, email, and salary.
- Song: title, artist, album, track number, . . .
- We can use one variable for each piece of information.
  - Or a list of records if we need multiple employees, songs, etc.
  - But it is much more convenient to group them together.
- In Python, we can group together these data into **objects**.
- An object has two components:
  - **Data** ("attributes", "members", "instance variables", "fields")
  - **Operations** that can be performed on the object ("methods", "member functions")
- We've already seen several kinds of objects:
  - Graphics shapes.
  - Strings.
  - Lists.
  - File objects.

Classes in Python

A **class** is a template for making many objects of the same kind.

Classes in Python

A **class** is a template for making many objects of the same kind.

- A class says what data the object contains,
  and what methods it supports.

Classes in Python

A **class** is a template for making many objects of the same kind.

- A class says what data the object contains, and what methods it supports.
  - All objects of the class have the same data variables and operations.
  - But each has its own *copy* of the data.

Classes in Python

A **class** is a template for making many objects of the same kind.

- A class says what data the object contains,
  and what methods it supports.
  - All objects of the class have the same data variables and operations.
  - But each has its own *copy* of the data.
- An object is called an **instance** of its class.
  - "Superhero" is a class, "Batman" an instance of Superhero.

Classes in Python

A **class** is a template for making many objects of the same kind.

- A class says what data the object contains,
  and what methods it supports.
  - All objects of the class have the same data variables and operations.
  - But each has its own *copy* of the data.
- An object is called an **instance** of its class.
  - "Superhero" is a class, "Batman" an instance of Superhero.
  - `Point` is a class, `Point(50,100)` an instance.

Classes in Python

A **class** is a template for making many objects of the same kind.

- A class says what data the object contains, and what methods it supports.
  - All objects of the class have the same data variables and operations.
  - But each has its own *copy* of the data.
- An object is called an **instance** of its class.
  - "Superhero" is a class, "Batman" an instance of Superhero.
  - `Point` is a class, `Point(50,100)` an instance.
  - `str` is a class, `"hello"` an instance.

Classes in Python

A **class** is a template for making many objects of the same kind.

- A class says what data the object contains, and what methods it supports.
  - All objects of the class have the same data variables and operations.
  - But each has its own *copy* of the data.
- An object is called an **instance** of its class.
  - "Superhero" is a class, "Batman" an instance of Superhero.
  - `Point` is a class, `Point(50,100)` an instance.
  - `str` is a class, `"hello"` an instance.
  - In Python: classes are types, objects are values.

Classes in Python

A **class** is a template for making many objects of the same kind.

- A class says what data the object contains,
  and what methods it supports.
  - All objects of the class have the same data variables and operations.
  - But each has its own *copy* of the data.
- An object is called an **instance** of its class.
  - "Superhero" is a class, "Batman" an instance of Superhero.
  - `Point` is a class, `Point(50,100)` an instance.
  - `str` is a class, `"hello"` an instance.
  - In Python: classes are types, objects are values.
- The class contains a **constructor** function that initializes new objects.

Classes in Python

A **class** is a template for making many objects of the same kind.

- A class says what data the object contains,
  and what methods it supports.
  - All objects of the class have the same data variables and operations.
  - But each has its own *copy* of the data.
- An object is called an **instance** of its class.
  - "Superhero" is a class, "Batman" an instance of Superhero.
  - `Point` is a class, `Point(50,100)` an instance.
  - `str` is a class, `"hello"` an instance.
  - In Python: classes are types, objects are values.
- The class contains a **constructor** function that initializes new objects.
  - Every time you call the constructor, you get a new object.
    - ★ With its own copy of the data.
    - ★ (whatever the constructor put there)

Classes in Python

A **class** is a template for making many objects of the same kind.

- A class says what data the object contains, and what methods it supports.
  - All objects of the class have the same data variables and operations.
  - But each has its own *copy* of the data.
- An object is called an **instance** of its class.
  - "Superhero" is a class, "Batman" an instance of Superhero.
  - `Point` is a class, `Point(50,100)` an instance.
  - `str` is a class, `"hello"` an instance.
  - In Python: classes are types, objects are values.
- The class contains a **constructor** function that initializes new objects.
  - Every time you call the constructor, you get a new object.
    - ⋆ With its own copy of the data.
    - ⋆ (whatever the constructor put there)

Class and constructor syntax

In Python you define a new class using the **class** keyword.

- The constructor is a function named `__init__` *inside* the class.
  - Two underscores before and two after!

Class and constructor syntax

In Python you define a new class using the **class** keyword.

- The constructor is a function named `__init__` *inside* the class.
  - Two underscores before and two after!

```
class Superhero:
    '''A specially talented purveyor of justice.'''
    def __init__(self, maskname):
        self.name = maskname
        self.identity = "unknown"
```

Class and constructor syntax

In Python you define a new class using the **class** keyword.

- The constructor is a function named `__init__` *inside* the class.
  - Two underscores before and two after!

```
class Superhero:
    '''A specially talented purveyor of justice.'''
    def __init__(self, maskname):
        self.name = maskname
        self.identity = "unknown"
```

- The constructor takes a special parameter `self`
  - This is the object that is being initialized.

Class and constructor syntax

In Python you define a new class using the **class** keyword.

- The constructor is a function named `__init__` *inside* the class.
  - Two underscores before and two after!

```
class Superhero:
    '''A specially talented purveyor of justice.'''
    def __init__(self, maskname):
        self.name = maskname
        self.identity = "unknown"
```

- The constructor takes a special parameter `self`
  - This is the object that is being initialized.
  - Can access the object's attributes with **dot notation**.

Class and constructor syntax

In Python you define a new class using the **class** keyword.

- The constructor is a function named `__init__` *inside* the class.
  - Two underscores before and two after!

```
class Superhero:
    '''A specially talented purveyor of justice.'''
    def __init__(self, maskname):
        self.name = maskname
        self.identity = "unknown"
```

- The constructor takes a special parameter `self`
  - This is the object that is being initialized.
  - Can access the object's attributes with **dot notation**.
  - Other parameters must be provided when calling the constructor: `darkknight = Superhero("Batman")`

Class and constructor syntax

In Python you define a new class using the **class** keyword.

- The constructor is a function named `__init__` *inside* the class.
  - Two underscores before and two after!

```
class Superhero:
    '''A specially talented purveyor of justice.'''
    def __init__(self, maskname):
        self.name = maskname
        self.identity = "unknown"
```

- The constructor takes a special parameter `self`
  - This is the object that is being initialized.
  - Can access the object's attributes with **dot notation**.
  - Other parameters must be provided when calling the constructor: `darkknight = Superhero("Batman")`
    - `maskname` will be "Batman".

Class and constructor syntax

In Python you define a new class using the **class** keyword.

- The constructor is a function named `__init__` *inside* the class.
    - Two underscores before and two after!

```
class Superhero:
    '''A specially talented purveyor of justice.'''
    def __init__(self, maskname):
        self.name = maskname
        self.identity = "unknown"
```

- The constructor takes a special parameter `self`
    - This is the object that is being initialized.
    - Can access the object's attributes with **dot notation**.
    - Other parameters must be provided when calling the constructor: `darkknight = Superhero("Batman")`
        - `maskname` will be "Batman".
        - Call the constructor using the *name of the class*, not `__init__`

Class and constructor syntax

In Python you define a new class using the **class** keyword.

- The constructor is a function named `__init__` *inside* the class.
  - Two underscores before and two after!

```
class Superhero:
    '''A specially talented purveyor of justice.'''
    def __init__(self, maskname):
        self.name = maskname
        self.identity = "unknown"
```

- The constructor takes a special parameter `self`
  - This is the object that is being initialized.
  - Can access the object's attributes with **dot notation**.
  - Other parameters must be provided when calling the constructor:
    `darkknight = Superhero("Batman")`
    - ★ `maskname` will be "Batman".
    - ★ Call the constructor using the *name of the class*, not `__init__`
    - ★ And leave out the "self" argument!

Class and constructor syntax

In Python you define a new class using the **class** keyword.

- The constructor is a function named `__init__` *inside* the class.
  - Two underscores before and two after!

```
class Superhero:
    '''A specially talented purveyor of justice.'''
    def __init__(self, maskname):
        self.name = maskname
        self.identity = "unknown"
```

- The constructor takes a special parameter `self`
  - This is the object that is being initialized.
  - Can access the object's attributes with **dot notation**.
  - Other parameters must be provided when calling the constructor: `darkknight = Superhero("Batman")`
    - ★ `maskname` will be "Batman".
    - ★ Call the constructor using the *name of the class*, not `__init__`
    - ★ And leave out the "self" argument!

Using a class

- You can create an object of the class by calling its constructor.
  - Use the name of the class as a function.
  - Arguments are the constructor parameters *but not `self`*.

Using a class

- You can create an object of the class by calling its constructor.
  - Use the name of the class as a function.
  - Arguments are the constructor parameters *but not `self`*.

    ```
    darkknight = Superhero("Batman")
    ```

Using a class

- You can create an object of the class by calling its constructor.
  - Use the name of the class as a function.
  - Arguments are the constructor parameters *but not* `self`.

```
darkknight = Superhero("Batman")
```

- You can access the object's attributes using dot notation.

```
print(darkknight.identity)
→unknown
```

Using a class

- You can create an object of the class by calling its constructor.
  - Use the name of the class as a function.
  - Arguments are the constructor parameters *but not `self`*.

    ```
    darkknight = Superhero("Batman")
    ```

- You can access the object's attributes using dot notation.

  ```
  print(darkknight.identity)
  →unknown
  print(darkknight.name)
  →Batman
  ```

Using a class

- You can create an object of the class by calling its constructor.
  - Use the name of the class as a function.
  - Arguments are the constructor parameters *but not `self`*.

    ```
    darkknight = Superhero("Batman")
    ```
- You can access the object's attributes using dot notation.

  ```
  print(darkknight.identity)
  →unknown
  print(darkknight.name)
  →Batman
  ```
- Can also **mutate** the object by assigning to an attribute.

Using a class

- You can create an object of the class by calling its constructor.
  - Use the name of the class as a function.
  - Arguments are the constructor parameters *but not self*.

    ```
    darkknight = Superhero("Batman")
    ```
- You can access the object's attributes using dot notation.

  ```
  print(darkknight.identity)
  →unknown
  print(darkknight.name)
  →Batman
  ```
- Can also **mutate** the object by assigning to an attribute.

  ```
  darkknight.identity = "Bruce Wayne"
  ```

Using a class

- You can create an object of the class by calling its constructor.
  - Use the name of the class as a function.
  - Arguments are the constructor parameters *but not `self`*.

    ```
    darkknight = Superhero("Batman")
    ```
- You can access the object's attributes using dot notation.

  ```
  print(darkknight.identity)
  →unknown
  print(darkknight.name)
  →Batman
  ```
- Can also **mutate** the object by assigning to an attribute.

  ```
  darkknight.identity = "Bruce Wayne"
  print(darkknight.identity)
  →Bruce Wayne
  ```

Using a class

- You can create an object of the class by calling its constructor.
  - Use the name of the class as a function.
  - Arguments are the constructor parameters *but not* `self`.

```
darkknight = Superhero("Batman")
```

- You can access the object's attributes using dot notation.

```
print(darkknight.identity)
→unknown
print(darkknight.name)
→Batman
```

- Can also **mutate** the object by assigning to an attribute.

```
darkknight.identity = "Bruce Wayne"
print(darkknight.identity)
→Bruce Wayne
```

- In **object-oriented programming** we often use methods instead of accessing attributes directly ("encapsulation").
  - `point.getX()`
  - `circle.setFill("blue")`

Using a class

- You can create an object of the class by calling its constructor.
  - Use the name of the class as a function.
  - Arguments are the constructor parameters *but not `self`*.

    ```
    darkknight = Superhero("Batman")
    ```

- You can access the object's attributes using dot notation.

  ```
  print(darkknight.identity)
  →unknown
  print(darkknight.name)
  →Batman
  ```

- Can also **mutate** the object by assigning to an attribute.

  ```
  darkknight.identity = "Bruce Wayne"
  print(darkknight.identity)
  →Bruce Wayne
  ```

- In **object-oriented programming** we often use methods instead of accessing attributes directly ("encapsulation").
  - `point.getX()`
  - `circle.setFill("blue")`

Defining methods

A **method** is a function defined in a class, operating on its instances.

Defining methods

A **method** is a function defined in a class, operating on its instances.

- **Object-oriented programming**: objects are data that can *do things*.

Defining methods

A **method** is a function defined in a class, operating on its instances.

- **Object-oriented programming**: objects are data that can *do things*.
- Methods are called using dot notation:
  `object.method(arguments)`

Defining methods

A **method** is a function defined in a class, operating on its instances.

- **Object-oriented programming**: objects are data that can *do things*.
- Methods are called using dot notation:
  `object.method(arguments)`
- Like the constructor, a method's first parameter is `self`.
  - This is the object on the left hand side of the dot.

Defining methods

A **method** is a function defined in a class, operating on its instances.

- **Object-oriented programming**: objects are data that can *do things*.
- Methods are called using dot notation: `object.method(arguments)`
- Like the constructor, a method's first parameter is `self`.
  - This is the object on the left hand side of the dot.
  - Other parameters correspond to method arguments.

Defining methods

A **method** is a function defined in a class, operating on its instances.

- **Object-oriented programming**: objects are data that can *do things*.
- Methods are called using dot notation:
  `object.method(arguments)`
- Like the constructor, a method's first parameter is `self`.
  - This is the object on the left hand side of the dot.
  - Other parameters correspond to method arguments.

```
class Superhero:
    ...
    def unmask(self, truename):
        self.identity = truename
        alert_villains(self.name + " is really "
                       + self.identity)
```

Defining methods

A **method** is a function defined in a class, operating on its instances.

- **Object-oriented programming**: objects are data that can *do things*.
- Methods are called using dot notation:
  `object.method(arguments)`
- Like the constructor, a method's first parameter is `self`.
  - This is the object on the left hand side of the dot.
  - Other parameters correspond to method arguments.

```
class Superhero:
   ...
   def unmask(self, truename):
       self.identity = truename
       alert_villains(self.name + " is really "
                      + self.identity)
```

```
darkknight.unmask("Bruce Wayne")
```

Defining methods

A **method** is a function defined in a class, operating on its instances.

- **Object-oriented programming**: objects are data that can *do things*.
- Methods are called using dot notation:
  `object.method(arguments)`
- Like the constructor, a method's first parameter is `self`.
  - This is the object on the left hand side of the dot.
  - Other parameters correspond to method arguments.

```
class Superhero:
    ...
    def unmask(self, truename):
        self.identity = truename
        alert_villains(self.name + " is really "
                       + self.identity)

darkknight.unmask("Bruce Wayne")
```

- `song.py`

Defining methods

A **method** is a function defined in a class, operating on its instances.

- **Object-oriented programming**: objects are data that can *do things*.
- Methods are called using dot notation:
  `object.method(arguments)`
- Like the constructor, a method's first parameter is `self`.
  - This is the object on the left hand side of the dot.
  - Other parameters correspond to method arguments.

```
class Superhero:
    ...
    def unmask(self, truename):
        self.identity = truename
        alert_villains(self.name + " is really "
                       + self.identity)
```

```
darkknight.unmask("Bruce Wayne")
```

- `song.py`

Encapsulation

Encapsulation means accessing an object only in a controlled way.

Encapsulation

Encapsulation means accessing an object only in a controlled way.

- Don't use the attributes directly.

Encapsulation

Encapsulation means accessing an object only in a controlled way.

- Don't use the attributes directly.
- Instead, use methods that allow only controlled changes.
- Why?

Encapsulation

Encapsulation means accessing an object only in a controlled way.

- Don't use the attributes directly.
- Instead, use methods that allow only controlled changes.
- Why?
  - In program 4, it is important that all the records have the same columns, in the same order.
  - All the functions/methods ensure that is the case...
  - ...but if you mess with the lists directly, all bets are off!

Encapsulation

Encapsulation means accessing an object only in a controlled way.

- Don't use the attributes directly.
- Instead, use methods that allow only controlled changes.
- Why?
  - In program 4, it is important that all the records have the same columns, in the same order.
  - All the functions/methods ensure that is the case...
  - ...but if you mess with the lists directly, all bets are off!
- Encapsulation allows our class to provide guarantees:

Encapsulation

Encapsulation means accessing an object only in a controlled way.

- Don't use the attributes directly.
- Instead, use methods that allow only controlled changes.
- Why?
  - In program 4, it is important that all the records have the same columns, in the same order.
  - All the functions/methods ensure that is the case. . .
  - . . . but if you mess with the lists directly, all bets are off!
- Encapsulation allows our class to provide guarantees:
  - A checked-in item should always have a shelf as its location.

Encapsulation

Encapsulation means accessing an object only in a controlled way.

- Don't use the attributes directly.
- Instead, use methods that allow only controlled changes.
- Why?
    - In program 4, it is important that all the records have the same columns, in the same order.
    - All the functions/methods ensure that is the case...
    - ...but if you mess with the lists directly, all bets are off!
- Encapsulation allows our class to provide guarantees:
    - A checked-in item should always have a shelf as its location.
    - Every domain has a matching IP address.

Encapsulation

Encapsulation means accessing an object only in a controlled way.

- Don't use the attributes directly.
- Instead, use methods that allow only controlled changes.
- Why?
  - In program 4, it is important that all the records have the same columns, in the same order.
  - All the functions/methods ensure that is the case...
  - ...but if you mess with the lists directly, all bets are off!
- Encapsulation allows our class to provide guarantees:
  - A checked-in item should always have a shelf as its location.
  - Every domain has a matching IP address.
  - Each employee is in the right tax bracket for their salary.

Encapsulation

Encapsulation means accessing an object only in a controlled way.

- Don't use the attributes directly.
- Instead, use methods that allow only controlled changes.
- Why?
  - In program 4, it is important that all the records have the same columns, in the same order.
  - All the functions/methods ensure that is the case. . .
  - . . . but if you mess with the lists directly, all bets are off!
- Encapsulation allows our class to provide guarantees:
  - A checked-in item should always have a shelf as its location.
  - Every domain has a matching IP address.
  - Each employee is in the right tax bracket for their salary.
  - Villains are notified when a superhero's identity is discovered.

Encapsulation

Encapsulation means accessing an object only in a controlled way.

- Don't use the attributes directly.
- Instead, use methods that allow only controlled changes.
- Why?
  - In program 4, it is important that all the records have the same columns, in the same order.
  - All the functions/methods ensure that is the case...
  - ...but if you mess with the lists directly, all bets are off!
- Encapsulation allows our class to provide guarantees:
  - A checked-in item should always have a shelf as its location.
  - Every domain has a matching IP address.
  - Each employee is in the right tax bracket for their salary.
  - Villains are notified when a superhero's identity is discovered.

C++

CS 215 uses C++, another programming language,

- Invented in early 80s by Bjarne Stroustrup at AT&T.

C++

CS 215 uses C++, another programming language,

- Invented in early 80s by Bjarne Stroustrup at AT&T.
  - Still actively developed.
  - Most recent update: C++14 (last year).

C++

CS 215 uses C++, another programming language,

- Invented in early 80s by Bjarne Stroustrup at AT&T.
  - Still actively developed.
  - Most recent update: C++14 (last year).
- Supports object-oriented programming like Python.
  - But is even better at encapsulation.

C++

CS 215 uses C++, another programming language,

- Invented in early 80s by Bjarne Stroustrup at AT&T.
  - Still actively developed.
  - Most recent update: C++14 (last year).
- Supports object-oriented programming like Python.
  - But is even better at encapsulation.
- Has an extensive built-in library
  - But nowhere near as extensive as Python's.

C++

CS 215 uses C++, another programming language,

- Invented in early 80s by Bjarne Stroustrup at AT&T.
  - Still actively developed.
  - Most recent update: C++14 (last year).
- Supports object-oriented programming like Python.
  - But is even better at encapsulation.
- Has an extensive built-in library
  - But nowhere near as extensive as Python's.
- Uses curly braces { } where Python requires indentation.
  - Many other pieces of syntax are different, too.

C++

CS 215 uses C++, another programming language,

- Invented in early 80s by Bjarne Stroustrup at AT&T.
  - Still actively developed.
  - Most recent update: C++14 (last year).
- Supports object-oriented programming like Python.
  - But is even better at encapsulation.
- Has an extensive built-in library
  - But nowhere near as extensive as Python's.
- Uses curly braces { } where Python requires indentation.
  - Many other pieces of syntax are different, too.
- **Compiled**, not interpreted.

C++

CS 215 uses C++, another programming language,

- Invented in early 80s by Bjarne Stroustrup at AT&T.
  - Still actively developed.
  - Most recent update: C++14 (last year).
- Supports object-oriented programming like Python.
  - But is even better at encapsulation.
- Has an extensive built-in library
  - But nowhere near as extensive as Python's.
- Uses curly braces { } where Python requires indentation.
  - Many other pieces of syntax are different, too.
- **Compiled**, not interpreted.
  - Translate the whole program into an executable file.
  - The user doesn't need C++ to run your program!

C++

CS 215 uses C++, another programming language,

- Invented in early 80s by Bjarne Stroustrup at AT&T.
  - Still actively developed.
  - Most recent update: C++14 (last year).
- Supports object-oriented programming like Python.
  - But is even better at encapsulation.
- Has an extensive built-in library
  - But nowhere near as extensive as Python's.
- Uses curly braces { } where Python requires indentation.
  - Many other pieces of syntax are different, too.
- **Compiled**, not interpreted.
  - Translate the whole program into an executable file.
  - The user doesn't need C++ to run your program!
  - This also means C++ programs can run much faster than Python.
    - ★ Because the translation has already been done.

C++

CS 215 uses C++, another programming language,

- Invented in early 80s by Bjarne Stroustrup at AT&T.
    - Still actively developed.
    - Most recent update: C++14 (last year).
- Supports object-oriented programming like Python.
    - But is even better at encapsulation.
- Has an extensive built-in library
    - But nowhere near as extensive as Python's.
- Uses curly braces { } where Python requires indentation.
    - Many other pieces of syntax are different, too.
- **Compiled**, not interpreted.
    - Translate the whole program into an executable file.
    - The user doesn't need C++ to run your program!
    - This also means C++ programs can run much faster than Python.
        - ★ Because the translation has already been done.
    - Other advantages?

C++

CS 215 uses C++, another programming language,

- Invented in early 80s by Bjarne Stroustrup at AT&T.
  - Still actively developed.
  - Most recent update: C++14 (last year).
- Supports object-oriented programming like Python.
  - But is even better at encapsulation.
- Has an extensive built-in library
  - But nowhere near as extensive as Python's.
- Uses curly braces { } where Python requires indentation.
  - Many other pieces of syntax are different, too.
- **Compiled**, not interpreted.
  - Translate the whole program into an executable file.
  - The user doesn't need C++ to run your program!
  - This also means C++ programs can run much faster than Python.
    - Because the translation has already been done.
  - Other advantages? Can keep the source code secret.

C++

CS 215 uses C++, another programming language,

- Invented in early 80s by Bjarne Stroustrup at AT&T.
  - Still actively developed.
  - Most recent update: C++14 (last year).
- Supports object-oriented programming like Python.
  - But is even better at encapsulation.
- Has an extensive built-in library
  - But nowhere near as extensive as Python's.
- Uses curly braces { } where Python requires indentation.
  - Many other pieces of syntax are different, too.
- **Compiled**, not interpreted.
  - Translate the whole program into an executable file.
  - The user doesn't need C++ to run your program!
  - This also means C++ programs can run much faster than Python.
    - ★ Because the translation has already been done.
  - Other advantages? Can keep the source code secret.

C++

- C++ is **statically typed** (Python is "dynamically typed")
  - You must specify the type when you create a variable.
    ```
    int score = 48;
    ```

C++

- C++ is **statically typed** (Python is “dynamically typed”)
  - You must specify the type when you create a variable.
    ```
    int score = 48;
    ```
  - The type never changes over the life of the variable.

C++

- C++ is **statically typed** (Python is "dynamically typed")
  - You must specify the type when you create a variable.
    ```
    int score = 48;
    ```
  - The type never changes over the life of the variable.
  - You also specify the types of parameters and return values.

C++

- C++ is **statically typed** (Python is "dynamically typed")
  - You must specify the type when you create a variable.
    ```
    int score = 48;
    ```
  - The type never changes over the life of the variable.
  - You also specify the types of parameters and return values.
  - Allows the compiler to detect many errors without running the code.

C++

- C++ is **statically typed** (Python is "dynamically typed")
  - You must specify the type when you create a variable.
    ```
    int score = 48;
    ```
  - The type never changes over the life of the variable.
  - You also specify the types of parameters and return values.
  - Allows the compiler to detect many errors without running the code.
- In general, Python is faster to write programs in.
  - The equivalent C++ program is often 3–10 times as long.

C++

- C++ is **statically typed** (Python is “dynamically typed”)
  - You must specify the type when you create a variable.
    `int score = 48;`
  - The type never changes over the life of the variable.
  - You also specify the types of parameters and return values.
  - Allows the compiler to detect many errors without running the code.
- In general, Python is faster to write programs in.
  - The equivalent C++ program is often 3–10 times as long.
- But C++ lets you write *faster programs*.
  - As much as 10 times as fast as the equivalent Python program.

C++

- C++ is **statically typed** (Python is "dynamically typed")
  - You must specify the type when you create a variable.
    `int score = 48;`
  - The type never changes over the life of the variable.
  - You also specify the types of parameters and return values.
  - Allows the compiler to detect many errors without running the code.
- In general, Python is faster to write programs in.
  - The equivalent C++ program is often 3–10 times as long.
- But C++ lets you write *faster programs*.
  - As much as 10 times as fast as the equivalent Python program.
- Sample C++ program: `cylinder.cpp`

C++

- C++ is **statically typed** (Python is “dynamically typed”)
  - You must specify the type when you create a variable.
    `int score = 48;`
  - The type never changes over the life of the variable.
  - You also specify the types of parameters and return values.
  - Allows the compiler to detect many errors without running the code.
- In general, Python is faster to write programs in.
  - The equivalent C++ program is often 3–10 times as long.
- But C++ lets you write *faster programs*.
  - As much as 10 times as fast as the equivalent Python program.
- Sample C++ program: `cylinder.cpp`