Summer Coding School 2023

Aug 26, 2023

Taehee Jeong, Ph.D.

Data type

- Integer
- Floating point
- String
- Boolean

Data structure

- List
- Set
- Tuples
- Dictionary

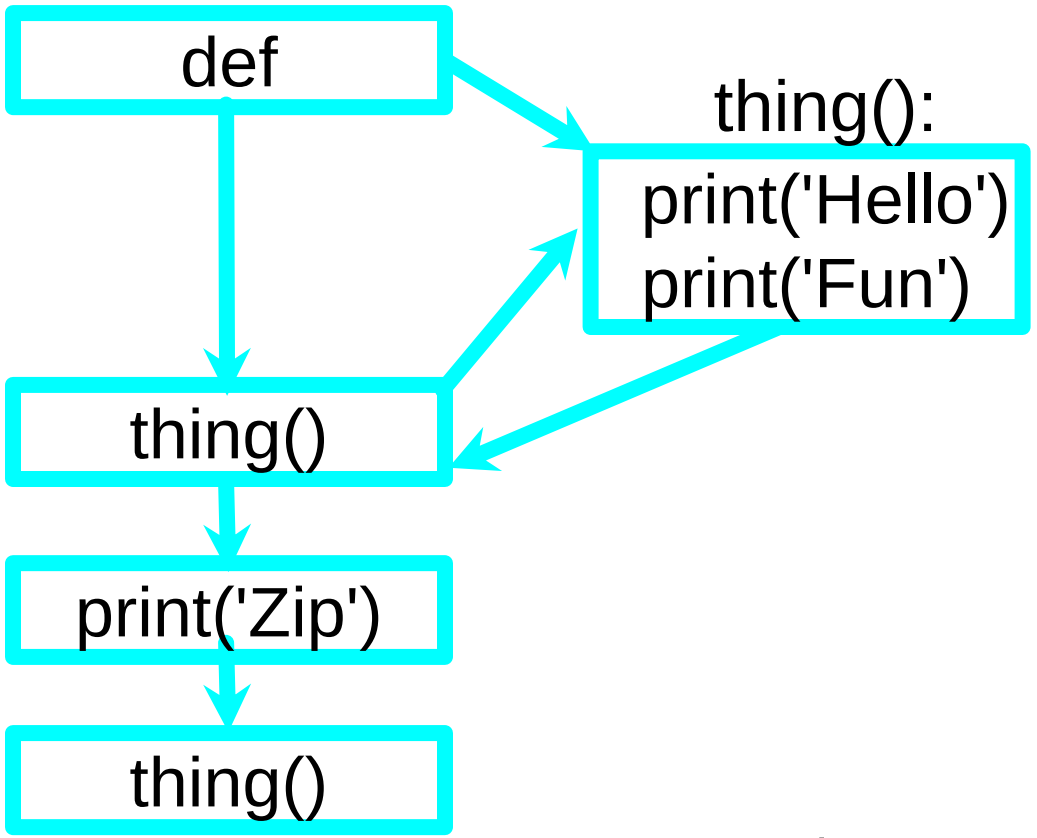
Control Flows

- Conditional: IF statement
- While Loop
- For Loop

Today’s Agenda

- Functions
- Class

Stored (and reused) Steps



Program:

```
def thing():
    print('Hello')
    print('Fun')

thing()
print('Zip')
thing()
```

Output:

```
Hello
Fun
Zip
Hello
Fun
```

We call these reusable pieces of code “functions”

Source: Python for Everybody, Charles Severance, University of Michigan School of Information

Python Functions

There are two kinds of functions in Python.

- Built-in functions that are provided as part of Python - print(), input(), type(), float(), int() ...

- Functions that we define ourselves and then use

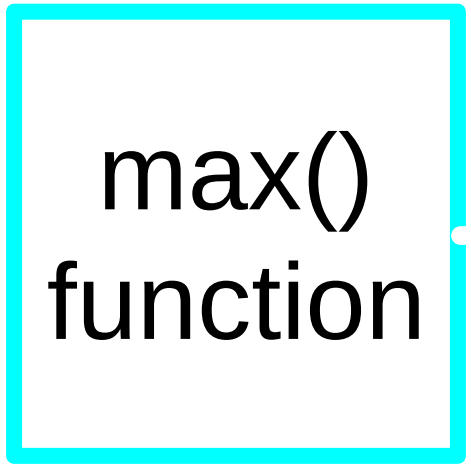
We treat the built-in function names as “new” reserved words
(i.e., we avoid them as variable names)

Max Function

```
>>> big = max('Hello world')
>>> print(big)
w
```

A function is some stored code that we use. A function takes some input and produces an output.

'Hello world'
(a string)



'w'
(a string)

Function Definition

In Python a function is some reusable code that takes arguments(s) as input, does some computation, and then returns a result or results

We define a function using the def reserved word

We call/invoke the function by using the function name, parentheses, and arguments in an expression

Source: Python for Everybody, Charles Severance, University of Michigan School of Information

Building our Own Functions

We create a new function using the def keyword followed by optional parameters in parentheses

We indent the body of the function

This defines the function but does not execute the body of the function

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')
```

Definitions and Uses

- Once we have defined a function, we can call (or invoke) it as many times as we like
- This is the store and reuse pattern

Arguments

- An argument is a value we pass into the function as its input when we call the function
- We use arguments so we can direct the function to do different kinds of work when we call it at different times
- We put the arguments in parentheses after the name of the function

```
big = max('Hello world')
```

Argument

Source: Python for Everybody, Charles Severance, University of Michigan School of Information

Parameters

A parameter is a variable which we use in the function definition.
It is a “handle” that allows the code in the function to access the arguments for a particular function invocation.

```
>>> def greet(lang):
...     if lang == 'es':
...         print('Hola')
...     elif lang == 'fr':
...         print('Bonjour')
...     else:
...         print('Hello')
...
>>> greet('en')
Hello
>>> greet('es')
Hola
>>> greet('fr')
Bonjour
```

Source: Python for Everybody, Charles Severance, University of Michigan School of Information

Return Values

Often a function will take its arguments, do some computation, and return a value to be used as the value of the function call in the calling expression.

```
def greet():
    return "Hello"

print(greet(), "Glenn")
print(greet(), "Sally")
```

```
Hello Glenn
Hello Sally
```

Return Value

A “fruitful” function is one that produces a result (or return value)

The return statement ends the function execution and “sends back” the result of the function

```
>>> def greet(lang):
...     if lang == 'es':
...         return 'Hola'
...     elif lang == 'fr':
...         return 'Bonjour'
...     else:
...         return 'Hello'
...
>>> print(greet('en'),'Glenn')
Hello Glenn
>>> print(greet('es'),'Sally')
Hola Sally
>>> print(greet('fr'),'Michael')
Bonjour Michael
```

Source: Python for Everybody, Charles Severance, University of Michigan School of Information

Multiple Parameters / Arguments

We can define more than one parameter in the function definition

We simply add more arguments when we call the function

We match the number and order of arguments and parameters

```
>>> def addtwo(a, b):
>>>     added = a + b
>>>     return added

>>> x = addtwo(3, 5)
>>> print(x)
>>> 8
```

Source: Python for Everybody, Charles Severance, University of Michigan School of Information

Void (non-fruitful) Functions

- When a function does not return a value, we call it a “void” function
- Functions that return values are “fruitful” functions
- Void functions are “not fruitful”

To function or not to function...

- Organize your code into “paragraphs” - capture a complete thought and “name it”
- Don’t repeat yourself - make it work once and then reuse it
- If something gets too long or complex, break it up into logical chunks and put those chunks in functions
- Make a library of common stuff that you do over and over - perhaps share this with your friends...

Source: Python for Everybody, Charles Severance, University of Michigan School of Information

A Class

class is a reserved word

Each PartyAnimal object has a bit of code

Tell the an object to run the party() code within it

```
class PartyAnimal:
   x = 0

   def party(self) :
     self.x = self.x + 1
     print("So far",self.x)

an = PartyAnimal()

an.party()
an.party()
an.party()
```

This is the template for making PartyAnimal objects

Each PartyAnimal object has a bit of data

Construct a PartyAnimal object and store in an

PartyAnimal.party(an)

Source: Python for Everybody, Charles Severance, University of Michigan School of Information

```
class PartyAnimal:
  x = 0

  def party(self) :
    self.x = self.x + 1
    print("So far",self.x)

an = PartyAnimal()

an.party()
an.party()
an.party()
```

```
$ python party1.py
So far 1
So far 2
So far 3
```

Source: Python for Everybody, Charles Severance, University of Michigan School of Information

Find Capabilities

- The dir() command lists capabilities
- Ignore the ones with underscores - these are used by Python itself
- The rest are real operations that the object can perform
- It is like type() - it tells us something *about* a variable

```
>>> y = list()
>>> type(y)
<class 'list'>
>>> dir(x)
['__add__', '__class__',
'__contains__', '__delattr__',
'__delitem__', '__delslice__',
'__doc__', … '__setitem__',
'__setslice__', '__str__', 'append',
'clear', 'copy', 'count', 'extend',
'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
```

Source: Python for Everybody, Charles Severance, University of Michigan School of Information

```
class PartyAnimal:
   x = 0

   def party(self) :
     self.x = self.x + 1
     print("So far",self.x)

an = PartyAnimal()

print("Type", type(an))
print("Dir ", dir(an))
```

We can use dir() to find the “capabilities” of our newly created class.

Source: Python for Everybody, Charles Severance, University of Michigan School of Information

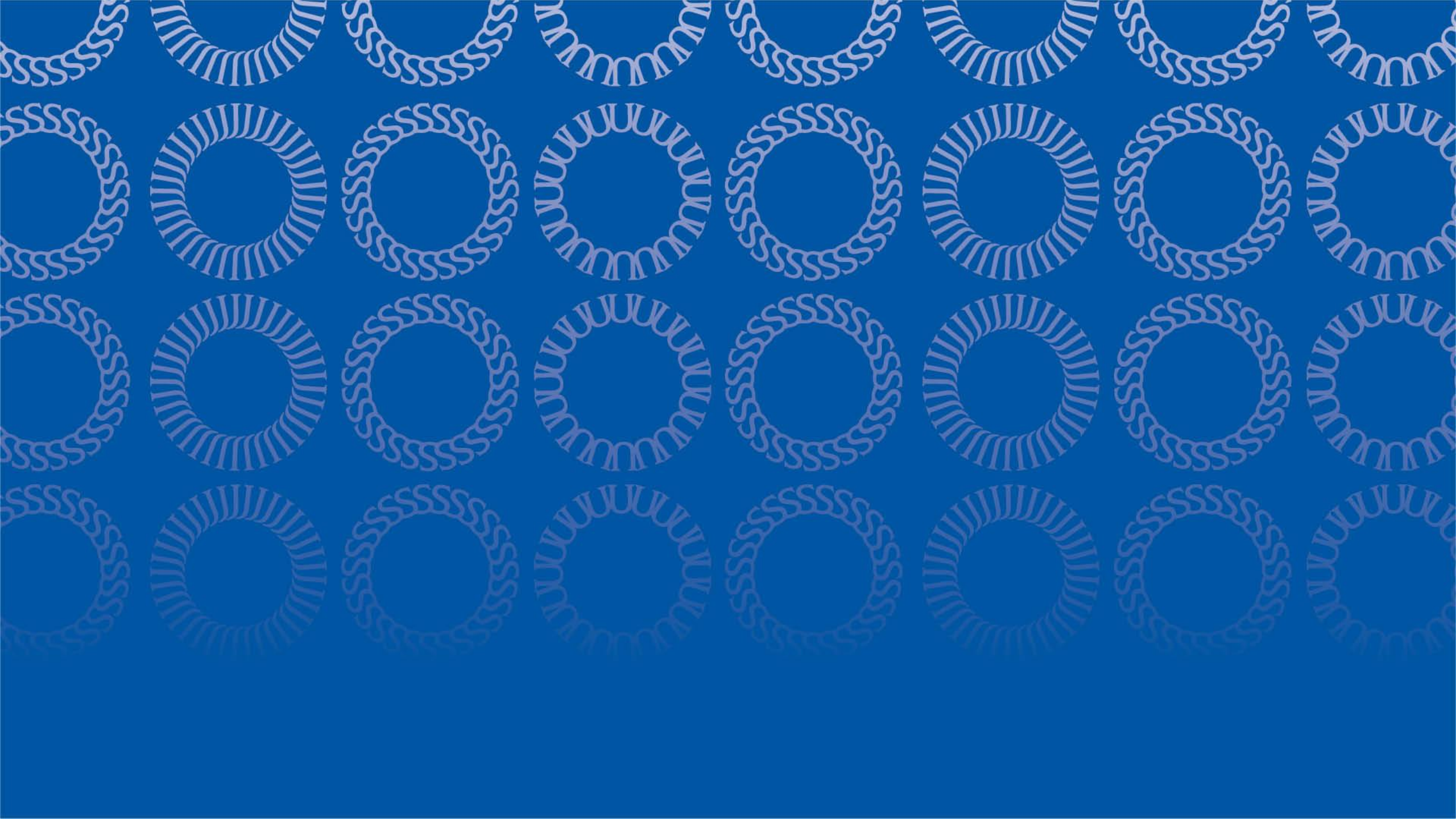
Try dir() with a String

```
>>> x = 'Hello there'
>>> dir(x)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__',
'__lt__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__str__',
'capitalize', 'center', 'count', 'decode', 'encode', 'endswith',
'expandtabs', 'find', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

Acknowledgements / Contributions

Initial Development: Charles Severance, University of Michigan School of Information

Modification: Taehee Jeong, San Jose State University



Object Oriented

A program is made up of many cooperating objects

Instead of being the “whole program” - each object is a little “island” within the program and cooperatively working with other objects

A program is made up of one or more objects working together - objects make use of each other’s capabilities

Object

An Object is a bit of self-contained Code and Data

A key aspect of the Object approach is to break the problem into smaller understandable parts (divide and conquer)

Objects have boundaries that allow us to ignore un-needed detail

We have been using objects all along: String Objects, Integer Objects, Dictionary Objects, List Objects...

Definitions

Class - a template

Method or Message - A defined capability of a class

Field or attribute- A bit of data in a class

Object or Instance - A particular instance of a class

Some Python Objects

```
>>> x = 'abc'
>>> type(x)
<class 'str'>
>>> type(2.5)
<class 'float'>
>>> type(2)
<class 'int'>
>>> y = list()
>>> type(y)
<class 'list'>
>>> z = dict()
>>> type(z)
<class 'dict'>
```

```
>>> dir(x)
[ … 'capitalize', 'casefold', 'center', 'count',
'encode', 'endswith', 'expandtabs', 'find',
'format', … 'lower', 'lstrip', 'maketrans',
'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper',
'zfill']
>>> dir(y)
[… 'append', 'clear', 'copy', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
>>> dir(z)
[…, 'clear', 'copy', 'fromkeys', 'get', 'items',
'keys', 'pop', 'popitem', 'setdefault', 'update',
'values']
```

Object Lifecycle

Objects are created, used, and discarded

We have special blocks of code (methods) that get called

- At the moment of creation (constructor)
- At the moment of destruction (destructor)

Constructors are used a lot

Destructors are seldom used

Constructor

The primary purpose of the constructor is to set up some instance variables to have the proper initial values when the object is created

Many Instances

We can create lots of objects - the class is the template for the object

We can store each distinct object in its own variable

We call this having multiple instances of the same class

Each instance has its own copy of the instance variables

```
class PartyAnimal:
   x = 0

   def __init__(self):
     print('I am constructed')

   def party(self) :
     self.x = self.x + 1
     print('So far',self.x)

   def __del__(self):
     print('I am destructed', self.x)

an = PartyAnimal()
an.party()
an.party()
an = 42
print('an contains',an)
```

The constructor and destructor are optional. The constructor is typically used to set up variables. The destructor is seldom used.

```
class PartyAnimal:
   x = 0
   name = ""
   def __init__(self, z):
     self.name = z
     print(self.name,"constructed")

   def party(self) :
     self.x = self.x + 1
     print(self.name,"party count",self.x)

s = PartyAnimal("Sally")
s.party()

j = PartyAnimal("Jim")
j.party()
s.party()
```

Constructors can have additional parameters. These can be used to set up instance variables for the particular instance of the class (i.e., for the particular object).

```
class PartyAnimal:
   x = 0
   name = ""
   def __init__(self, z):
     self.name = z
     print(self.name,"constructed")

   def party(self) :
     self.x = self.x + 1
     print(self.name,"party count",self.x)

s = PartyAnimal("Sally")
s.party()

j = PartyAnimal("Jim")
j.party()
s.party()
```

We have two independent instances

Inheritance

When we make a new class - we can reuse an existing class and inherit all the capabilities of an existing class and then add our own little bit to make our new class

Another form of store and reuse

Write once - reuse many times

The new class (child) has all the capabilities of the old class (parent) - and then some more

```
class PartyAnimal:
   x = 0
   name = ""
   def __init__(self, nam):
     self.name = nam
     print(self.name,"constructed")

   def party(self) :
     self.x = self.x + 1
     print(self.name,"party count",self.x)

class FootballFan(PartyAnimal):
   points = 0
   def touchdown(self):
      self.points = self.points + 7
      self.party()
      print(self.name,"points",self.points)
```

```
s = PartyAnimal("Sally")
s.party()

j = FootballFan("Jim")
j.party()
j.touchdown()
```

FootballFan is a class which extends PartyAnimal. It has all the capabilities of PartyAnimal and more.

Definitions

Class - a template

Attribute – A variable within a class

Method - A function within a class

Object - A particular instance of a class

Constructor – Code that runs when an object is created

Inheritance - The ability to extend a class to make a new class.