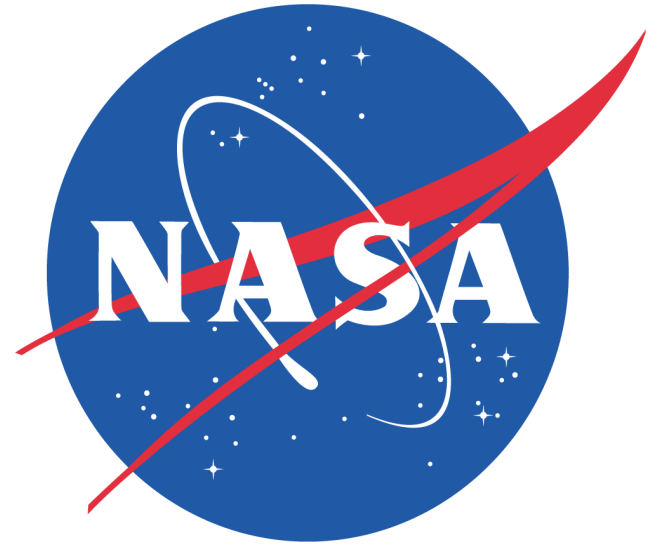




Python!

- Created in 1991 by Guido van Rossum
 - Named for Monty Python
- Useful as a **scripting language**
 - **script**: A small program meant for one-time use
 - Targeted towards small to medium sized projects
- Used by:
 - Google, Yahoo!, Youtube
 - Many Linux distributions
 - Games and apps (e.g. Eve Online)

Python is used everywhere!



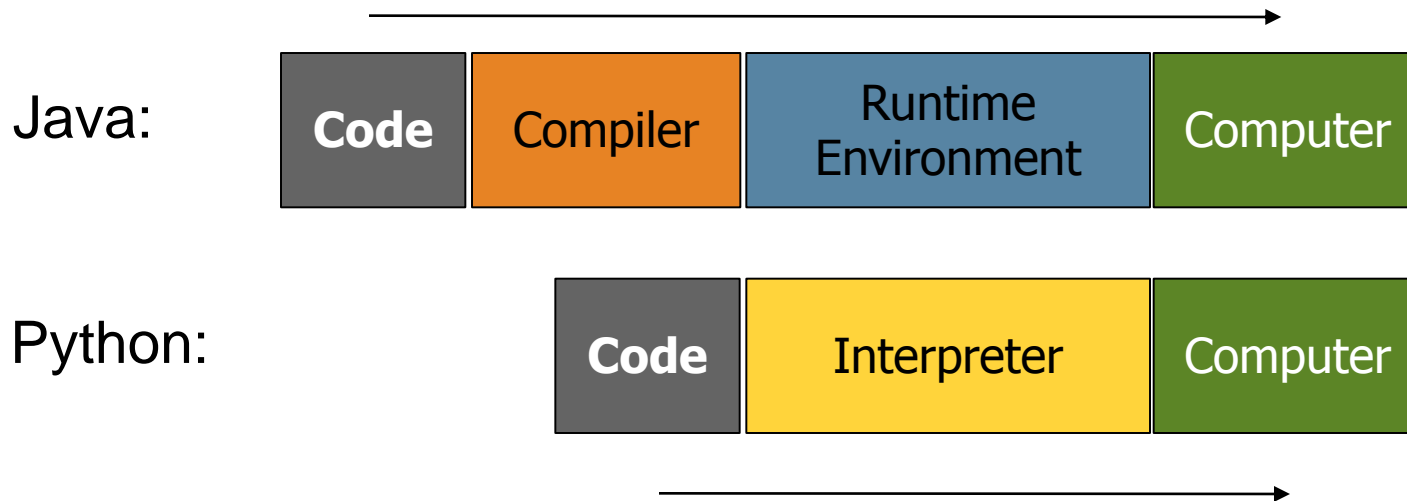
About Python

- Easy to learn, powerful programming language.
 - Website development, data analysis, server maintenance, numerical analysis, image processing,....
- Elegant syntax
- It has efficient high-level data structures and a simple but effective approach to object-oriented programming.
- Comprehensive standard library for many tasks
- Big community
- Extensible, embeddable
- Portable
- Interpreted

Interpreted Languages

- **Interpreted**

- Not compiled like Java
- Code is written and then directly executed by an **interpreter**
- Type commands into interpreter and see immediate results



History

- Start implementation in December 1989 by Guido van Rossum
- First released in 1991
- 16.10.2000: Python 2.0
- 3.12.2008: Python 3.0
- Current version: Python 2.7.15 and Python 3.7.2

Zen of Python

Software principles that influence the design of Python.

(import this)

- 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.
- Special cases aren't special enough to break the rules.
- Errors should never pass silently.

• Unless explicitly silenced.



Installing Python

Windows:

- Download Python from <http://www.python.org>
- Install Python.
- Run **Idle** from the Start Menu.

Mac OS X:

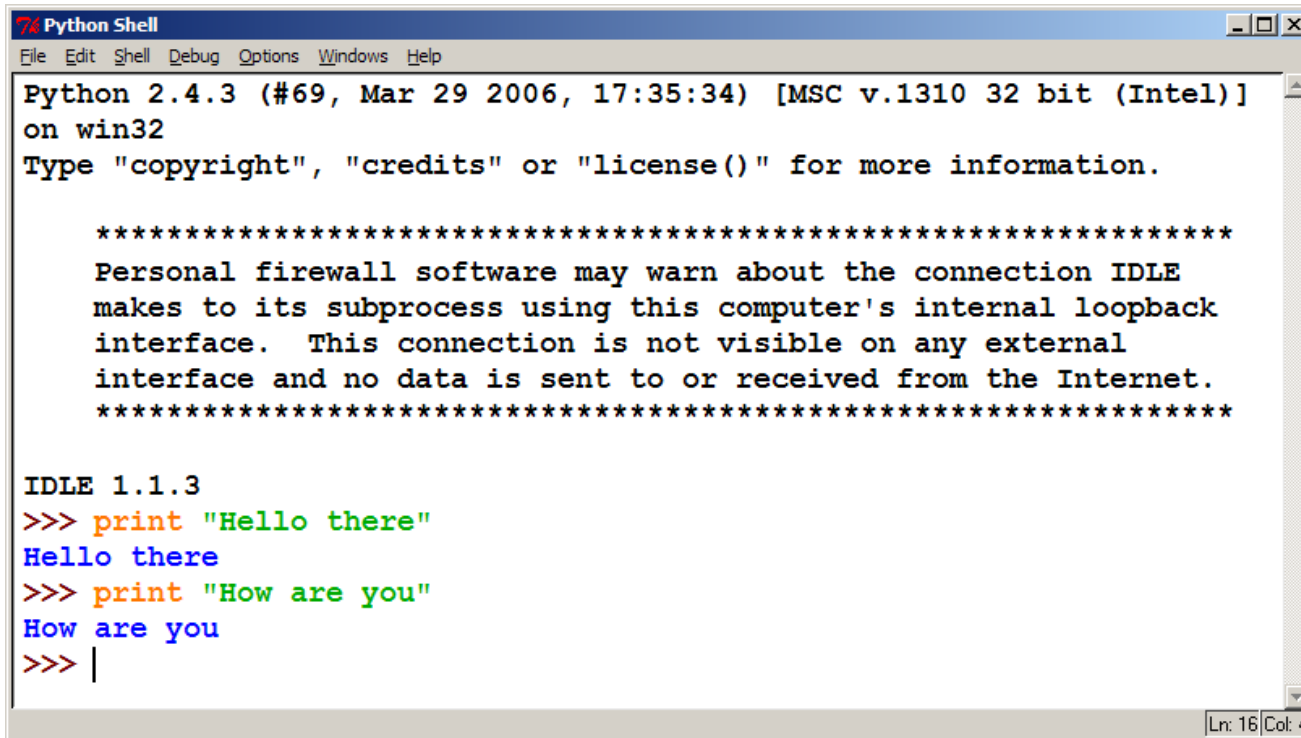
- Python is already installed.
- Open a terminal and run `python` or run Idle from Finder.

Linux:

- Chances are you already have Python installed. To check, run `python` from the terminal.
- If not, install from your distribution's package system.

The Python Interpreter

- Allows you to type commands one-at-a-time and see results
- A great way to explore Python's syntax
 - Repeat previous command: *Alt+P*



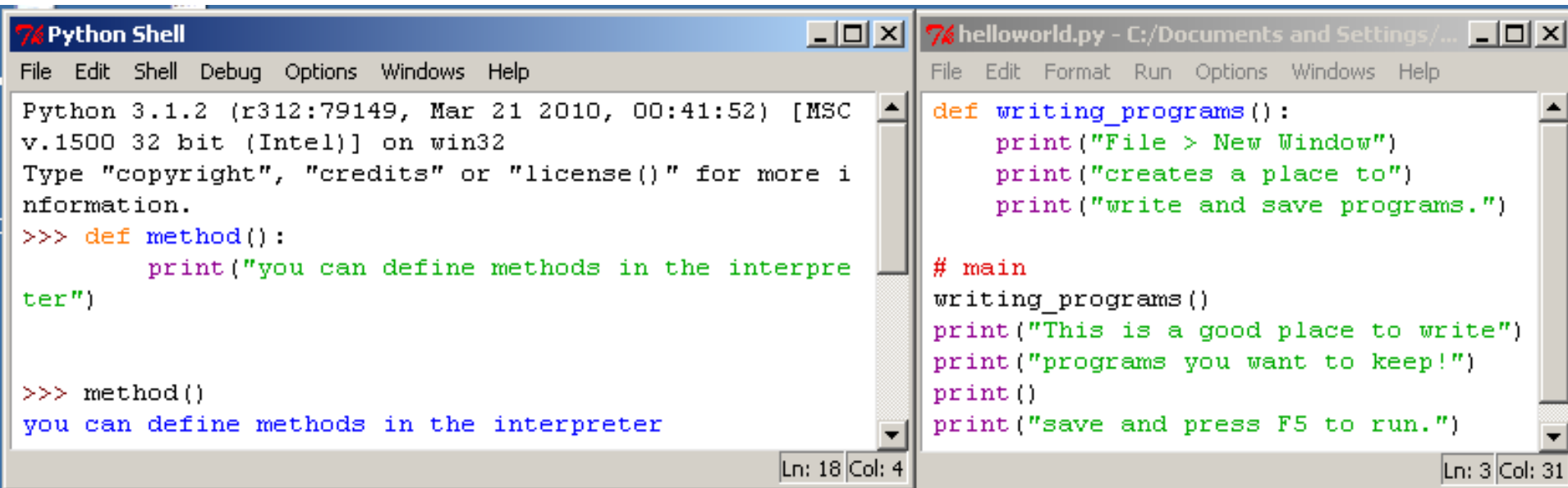
```
Python Shell
File Edit Shell Debug Options Windows Help
Python 2.4.3 (#69, Mar 29 2006, 17:35:34) [MSC v.1310 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.

*****
Personal firewall software may warn about the connection IDLE
makes to its subprocess using this computer's internal loopback
interface. This connection is not visible on any external
interface and no data is sent to or received from the Internet.
*****

IDLE 1.1.3
>>> print "Hello there"
Hello there
>>> print "How are you"
How are you
>>> |
```

How to run Python Windows

- Run IDLE to use the interpreter
- Open a new window in IDLE to write and save programs



The screenshot displays the Python IDLE (Integrated Development and Learning Environment) interface. It consists of two main windows:

- Python Shell:** This window is on the left. It shows the Python 3.1.2 interpreter running on Windows. The prompt is `>>>`. A user-defined function `method()` has been created, which prints the message "you can define methods in the interpreter". The function has been called, and the output is displayed.
- helloworld.py:** This window is on the right. It contains a Python script with a function `writing_programs()` that prints three lines of text. Below the function, there is a `# main` section that calls `writing_programs()` and prints additional text.

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.1.2 (r312:79149, Mar 21 2010, 00:41:52) [MSC
v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more i
nformation.
>>> def method():
    print("you can define methods in the interpre
ter")
>>> method()
you can define methods in the interpreter
Ln: 18 Col: 4
```

```
helloworld.py - C:/Documents and Settings/...
File Edit Format Run Options Windows Help
def writing_programs():
    print("File > New Window")
    print("creates a place to")
    print("write and save programs.")
# main
writing_programs()
print("This is a good place to write")
print("programs you want to keep!")
print()
print("save and press F5 to run.")
Ln: 3 Col: 31
```

Hello, world! in JAVA

- Console output: `System.out.println()` ;
- Methods: `public static void name() { ... }` □

Hello2.java

```
1 public class Hello2 {  
2     public static void main(String[] args) {  
3         hello();  
4     }  
5  
6     public static void hello() {  
7         System.out.println("Hello, world!");  
8     }  
9 }
```

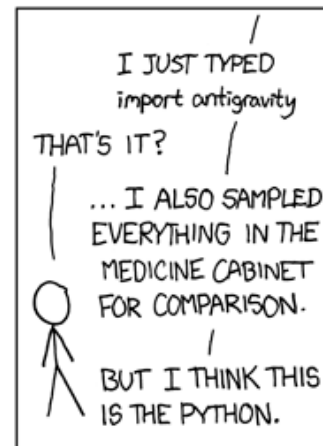
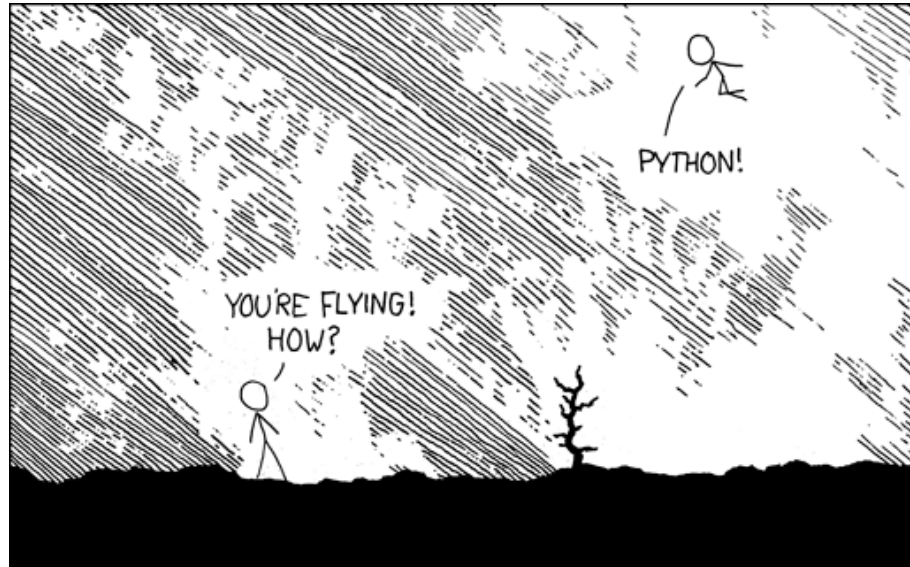
Our First Python Program

- Python does not have a `main` method like Java
 - The program's main code is just written directly in the file
- Python statements do not end with semicolons

hello.py

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

A Brief Review



Python 2.x vs Python 3.x

- We will be using Python 3 for this course
- Sometimes we may refer to Python 2
- The differences are minimal

How to	Python 2.x	Python 3.x
print text	<code>print "text"</code>	<code>print("text")</code>
print a blank line	<code>print</code>	<code>print()</code>

Documentation

Online help in the interpreter:

- **help()**: general Python help
- **help(obj)**: help regarding an object, e.g. a function or a module
- **dir ()** : all used names
- **dir(obj)**: all attributes of an object

Official documentation: <http://docs.python.org/>

The `print` Statement

```
print("text")  
print() (a blank line)
```

- Escape sequences such as `\` are the same as in Java
- Strings can also start/end with `'`

swallows.py

```
1 print("Hello, world! ")  
2 print()  
3 print("Suppose two swallows \"carry\" it together.")  
4 print('African or "European" swallows?')
```


Comments

comment text (one line)

must start each line of comments with the pound sign

swallows2.py

```
1  # This program prints important messages.
2  print("Hello, world!")
3  print()                    # blank line
4  print("Suppose two swallows \"carry\" it together.")
5  Print('African or "European" swallows?')
6
```

Functions

- **Function:** Equivalent to a static method in Java.
- **Syntax:**

```
def name () :  
    statement  
    statement  
    ...  
    statement
```

hello2.py

```
1  # Prints a helpful message.  
2  def hello():  
3      print("Hello, world!")  
4  
5  # main (calls hello twice)  
6  hello()  
7  hello()
```

- Must be declared above the 'main' code
- Statements inside the function must be indented

Whitespace Significance

- Python uses indentation to indicate blocks, instead of { }
 - Makes the code simpler and more readable
 - In Java, indenting is optional. In Python, you **must** indent.
 - You may use either tabs or spaces, but you **must** be consistent. Statements which go together must have the same indentation level.

hello3.py

```
1  # Prints a welcoming message.
2  def hello():
3      ➡ print("Hello, world!")
4      ➡ print("How are you?")
5
6  # main (calls hello twice)
7  hello()
8  hello()
```

Tabs or Spaces

shell

```
1  # Prints a helpful message.
2  >>> def indent_reminder():
3  ...   print("Remember, you must indent!")
4  ...   File "<stdin>", line 2
5  ...       print("Remember, you must indent!")
6  ...       ^
7  IndentationError: expected an indented block
8  >>> def indent_reminder():
9  ...     print("Remember, you must indent!")
10 ...
11 >>> indent_reminder()
12 Remember, you must indent!
>>>
```

Tabs or Spaces

shell

```
1  # Prints a helpful message.
2  >>> def indentation_errors():
3  ...     print("this was indented using a tab")
4  ...     print("this was indented using four spaces")
5  File "<stdin>", line 3
6  ...     print("this was indented using four spaces")
7  ^
8  IndentationError: unindent does not match any outer
9  indentation level
10 >>> def indentation_errors():
11 ...     print("this was indented using a tab")
12 ...     print("so this must also use a tab")
...
>>> def more_indentation_tricks():
...     print("If I use spaces to indent here.")
...     print("then I must use spaces to indent here.")
>>>
```

Variables

A variable is a name that refers to a value. An assignment statement creates new variables and gives them values.

- Declaring
 - no type is written; same syntax as assignment
- Operators
 - no ++ or -- operators (must manually adjust by 1)

Java	Python
<pre>int x = 2; x++; System.out.println(x); x = x * 8; System.out.println(x); double d = 3.2; d = d / 2; System.out.println(d);</pre>	<pre>x = 2 x = x + 1 print(x) x = x * 8 print(x) d = 3.2 d = d / 2 print(d)</pre>

Types

- Python is looser about types than Java
 - Variables' types do not need to be declared
 - Variables can change types as a program is running

Value	Java type	Python type
42	int	int
3.14	double	float
"ni!"	String	str

- There is no separate long type. The int can be of any size.
- You can find the type of anything using the `type()` function

Variable Names & Keywords

- Can be arbitrary long
- Can contain both letters and numbers, but cannot start with a number. Legal to use upper case letters
- `_` underscore character can appear in a name

Python reserves 33 keywords:

`and, def, from, None, True, as, elif, try, pass, del,`

They cannot be used as variable names.

Operators on Numbers

- Arithmetic is very similar to Java
 - Operators: `+` `-` `*` `/`
 - **Div and modulo**: `//`, `%`, `divmod(x, y)`
 - **Power**: `x**y`, `pow(x, y)`
 - Precedence: `()` before `**` before `*` `/` `%` before `+` `-`
 - Integers vs. real numbers (doubles)
 - You may use `//` for integer division
 - Absolute value: `abs(x)`
 - Rounding: `round(x)`
 - Conversion: `int(x)`, `float(x)`

```
>>> 1 + 1
2
>>> 1 + 3 * 4 - 2
11
>>> 7 // 2
3
>>> 7 / 2
3.5
>>> 7.0 / 2
3.5
```

Strings

A string is a sequence of characters.

- Data type: str
- `s='spam'`, `s="spam"` (single and double quotes: same)
- Multiline strings: using triple quotes `"""spam"""` or `'''spam'''`
- Strings are immutable: once you have created a string you cannot change it.
- Raw string: `r"sp\nam"`

```
>>>print("sp\nam")
sp
am
>>> print(r"sp\nam")
Sp\nam
>>>s= '''hello
... World'''
>>>print(s)
```



String Multiplication

- Python strings can be multiplied by an integer.
 - The result is many copies of the string concatenated together.

```
>>> "hello" * 3
"hellohellohello"

>>> print(10 * "yo ")
yo yo yo yo yo yo yo yo yo yo

>>> print(2 * 3 * "4")
444444
```

String Concatenation

- Integers and strings cannot be concatenated in Python.
 - Workarounds:
 - `str(value)` - converts a value into a string
 - `print(value, value)` - prints values, separated by a space

```
>>> x = 4
>>> print("Thou shalt not count to " + x + ".")
TypeError: cannot concatenate 'str' and 'int' objects

>>> print("Thou shalt not count to " + str(x) + ".")
Thou shalt not count to 4.

>>> print(x + 1, "is out of the question.")
5 is out of the question.
```

Special Print Options

- You may define the behavior of the print function as follows:
 - `end`: string appended after the last value, default `\n`
 - `sep`: string inserted between values, default a space

```
>>> print("One", "two", "five", sep="... ")
One... two... five

>>> print("O", "M", "G", sep="...   ", end="!!!1")
O...   M...   G!!!1
```

for Loop Variations

- `for name in range(min, max) :`
- `statements`
- `for name in range(min, max, step) :`
- `statements`
- Can specify a minimum other than 0, and a step other than 1

```
>>> for i in range(2, 6):  
...     print(i)  
2  
3  
4  
5  
>>> for i in range(15, 0, -5):  
...     print(i)  
15  
10  
5
```

Nested Loops

- Nested loops are often replaced by string `*` and `+`

....1
...2
..3
.4
5

Java

```
1 for (int line = 1; line <= 5; line++) {  
2     for (int j = 1; j <= (5 - line); j++) {  
3         System.out.print(".");  
4     }  
5     System.out.println(line);  
6 }
```

Python

```
1 for line in range(1, 6):  
2     print((5 - line) * ".", line, sep="")
```

Constants

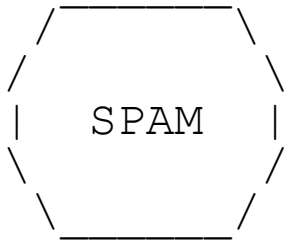
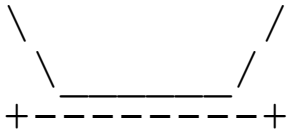
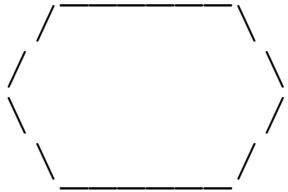
- Python doesn't really have constants.
 - Instead, declare a variable at the top of your code.
 - All methods will be able to use this "constant" value.

constant.py

```
1 MAX_VALUE = 3
2
3 def print_top():
4     for i in range(MAX_VALUE):
5         for j in range(i):
6             print(j)
7         print()
8
9 def print_bottom():
10    for i in range(MAX_VALUE, 0, -1):
11        for j in range(i, 0, -1):
12            print(MAX_VALUE)
13        print()
```


Exercise

- Write a program in Python that gives the following output:



Exercise Solution

```
def egg():  
    top()  
    bottom()  
    print()
```

```
def cup():  
    bottom()  
    line()  
    print()
```

```
def stop():  
    top()  
    print("|   SPAM   |")  
    bottom()  
    print()
```

```
def hat():  
    top()  
    line()  
    print()
```

```
def top():  
    print("           ")  
    print(" /_____\\"")  
    print("/          \\"")
```

```
def bottom():  
    print("\\           /")  
    print("  \\_____/" )
```

```
def line():  
    print("+-----+")
```

```
# main  
egg()  
cup()  
stop()  
hat()
```

Exercise

- Write a code in Python that gives the following output:

```
#=====#
|          <><>          |
|      <> . . . . <>      |
|  <> . . . . . <>  |
| <> . . . . . <> |
| <> . . . . . <> |
|  <> . . . . . <>  |
|      <> . . . . <>      |
|          <><>          |
#=====#
```

- Make the mirror resizable by using a "constant."

```

def bar():
    print("#", 16 * "=", "#")

def draw_top_half():
    for line in range(1, 5):
        print("|", end=" ")
        print(" " * (-2 * line + 8), end="<>")
        print("." * (4 * line - 4), end="<>")
        print(" " * (-2 * line + 8), end=" ")
        print("|", end="\n")

def draw_bottom_half():
    for line in range(4, 0, -1):
        print("|", end=" ")
        print(" " * (-2 * line + 8), end="<>")
        print("." * (4 * line - 4), end="<>")
        print(" " * (-2 * line + 8), end=" ")
        print("|", end="\n")

bar()
draw_top_half()
draw_bottom_half()
bar()

```

Exercise Solution

```
# constant
SIZE = 4

def bar():
    print("#", 4 * SIZE * "=", "#")

def draw_top_half():
    for line in range(1, SIZE + 1):
        print("|", end=" ")
        print(" " * (-2 * line + 2 * SIZE), end="<>")
        print("." * (4 * line - 4), end="<>")
        print(" " * (-2 * line + 8), end=" ")
        print("|", end="\n")

def draw_bottom_half():
    for line in range(SIZE, 0, -1):
        print("|", end=" ")
        print(" " * (-2 * line + 2 * SIZE), end="<>")
        print("." * (4 * line - 4), end="<>")
        print(" " * (-2 * line + 2 * SIZE), end=" ")
        print("|", end="\n")

bar()
draw_top_half()
draw_bottom_half()
bar()
```

Concatenating Ranges

- Ranges can be concatenated with +
 - However, you must use the “list()” command
 - Can be used to loop over a disjoint range of numbers

```
>>> list(range(1, 5)) + list(range(10, 15))
[1, 2, 3, 4, 10, 11, 12, 13, 14]

>>> for i in list(range(4)) + list(range(10, 7, -1)):
...     print(i)
0
1
2
3
10
9
8
```

Exercise Solution 2

```
SIZE = 4
```

```
def bar():  
    print("#", 4 * SIZE * "=", "#")
```

```
def mirror():  
    for line in list(range(1, SIZE + 1)) + list(range(SIZE, 0, -1)):  
        print("|", end=" ")  
        print(" " * (-2 * line + 2 * SIZE), end="<>")  
        print("." * (4 * line - 4), end="<>")  
        print(" " * (-2 * line + 8), end=" ")  
        print("|", end="\n")
```

```
# main  
bar()  
mirror()  
bar()
```

input

- `input` : Reads a string from the user's keyboard.
- reads and returns an entire line of input

```
>>> name = input("Howdy. What's your name? ")
Howdy. What's your name? Virat Kohli

>>> name
'Virat Kohli'
```


input for numbers

- to read a number, cast the result of `input` to an `int`
 - Only numbers can be cast as `int`!
 - Example:

```
age = int(input("How old are you? "))  
print("Your age is", age)  
print("You have", 65 - age, "years until  
retirement")
```

Output:

```
How old are you? 53  
Your age is 53  
You have 12 years until retirement
```

if

if condition:
statements

– Example:

```
gpa = int(input("What is your GPA? "))  
if gpa > 6.5:  
    print("Your application is accepted.")
```

if/else

```
if condition:  
    statements  
elif condition:  
    statements  
else:  
    statements
```

– Example:

```
gpa = int(input("What is your GPA? "))  
if gpa > 7:  
    print("You have qualified for the honor roll.")  
elif gpa > 5.0:  
    print("Welcome to Moon University!")  
else:  
    print("Apply again in Spring.")
```

if ... in

if value in sequence:
statements

- The sequence can be a range, string, tuple, or list
- Examples:

```
x = 3
```

```
if x in range(0, 10):  
    print("x is between 0 and 9")
```

```
name = input("What is your name? ")  
name = name.lower()
```

```
if name[0] in "aeiou":  
    print("Your name starts with a vowel!")
```

Comparison and Logical Operators

Operator	Meaning	Example	Result
<code>==</code>	equals	<code>1 + 1 == 2</code>	True
<code>!=</code>	does not equal	<code>3.2 != 2.5</code>	True
<code><</code>	less than	<code>10 < 5</code>	False
<code>></code>	greater than	<code>10 > 5</code>	True
<code><=</code>	less than or equal to	<code>126 <= 100</code>	False
<code>>=</code>	greater than or equal to	<code>5.0 >= 5.0</code>	True

Operator	Example	Result
<code>and</code>	<code>(2 == 3) and (-1 < 5)</code>	False
<code>or</code>	<code>(2 == 3) or (-1 < 5)</code>	True
<code>not</code>	<code>not (2 == 3)</code>	True

Functions: Parameters

```
def name(parameter, parameter, ..., parameter) :  
    statements
```

- Parameters are declared by writing their names (no types)

```
>>> def print_many(word, n) :  
...     for i in range(n):  
...         print(word)  
  
>>> print_many("hello", 4)  
hello  
hello  
hello  
hello
```

Exercise

- Create the lines/boxes of stars as follows:

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*           *
```

```
*****
```

```
*****
```

```
*           *
```

```
*           *
```

```
*****
```

Exercise Solution

stars.py

```
1  # Draws a box of stars with the given width and height.
2  def box(width, height):
3      print width * "*"
4      for i in range(height - 2):
5          print("*" + (width - 2) * " " + "*")
6      print width * "*"
7
8  # main
9  print(13 * "*")
10 print(7 * "*")
11 print(35 * "*")
12 box(10, 3)
13 box(5, 4)
```


Default Parameter Values

def **name** (**parameter=value**, ..., **parameter=value**) :
 statements

- Can make parameter(s) optional by specifying a default value

```
>>> def print_many(word, n=1):  
...     for i in range(n):  
...         print(word)
```

```
>>> print_many("shrubbery")  
shrubbery  
>>> print_many("shrubbery", 4)  
shrubbery  
shrubbery  
shrubbery  
shrubbery
```

- **Exercise:** Modify `stars.py` to add an optional parameter for the character to use for the outline of the box (default "*").

Parameter Keywords

def **name** (**parameter=value**, ..., **parameter=value**)

- Can specify the names of parameters as you call a function
- This allows you to pass the parameters in any order

```
>>> def print_many(word, n):  
...     for i in range(n):  
...         print(word)  
  
>>> print_many(word="shrubbery", n=4)  
shrubbery  
shrubbery  
shrubbery  
shrubbery  
>>> print_many(n=3, word="Ni!")  
Ni!  
Ni!  
Ni!
```

Returning Values

```
def name (parameters) :  
    statements  
    ...  
    return value
```

```
>>> def ftoc(temp):  
...     tempc = 5.0 / 9.0 * (temp - 32) #Fahrenheit to Cels.  
...     return tempc  
  
>>> ftoc(98.6)  
37.0
```

- **Functions accept arbitrary objects as parameters and return values.**
- **Functions without explicit return value return **None****

Functions and Modules

- Functions thematically belonging together can be stored in a separate Python file. (Same for objects and classes)
- This file is called module and can be loaded in any Python script.
- Multiple modules available in the Python Standard Library (part of the Python installation)
- Command for loading a module: `import <filename>`
(filename without ending `.py`)

```
>>>import math
```

```
>>>s = math.sin(math.pi)
```

Math Functions

`from math import *` `or import math` `then math.*`

Function name	Description
<code>ceil(value)</code>	rounds up
<code>cos(value)</code>	cosine, in radians
<code>degrees(value)</code>	convert radians to degrees
<code>floor(value)</code>	rounds down
<code>log(value, base)</code>	logarithm in any base
<code>log10(value)</code>	logarithm, base 10
<code>max(value1, value2, ...)</code>	largest of two (or more) values
<code>min(value1, value2, ...)</code>	smallest of two (or more) values
<code>radians(value)</code>	convert degrees to radians
<code>round(value)</code>	nearest whole number
<code>sin(value)</code>	sine, in radians
<code>sqrt(value)</code>	square root
<code>tan(value)</code>	tangent

Constant	Description
<code>e</code>	2.7182818...
<code>pi</code>	3.1415926...

Strings

index	0	1	2	3	4	5	6	7
<i>or</i>	-8	-7	-6	-5	-4	-3	-2	-1
character	P	.		D	i	d	d	y

- Accessing character(s):
 variable [**index**]
 variable [**index1:index2**]
 - **index2** is exclusive
 - **index1** or **index2** can be omitted (end of string)

```
>>> name = "P. Diddy"  
>>> name[0]  
'P'  
>>> name[7]  
'y'  
>>> name[-1]  
'y'  
>>> name[3:6]  
'Did'  
>>> name[3:]  
'Diddy'  
>>> name[:-2]  
'P. Did'
```

String Methods

Java	Python
length	len(str)
startsWith, endsWith	startswith, endswith
toLowerCase, toUpperCase	upper, lower, isupper, islower, capitalize, swapcase
indexOf	find
trim	strip

```
>>> name = "Jordan Hiroshi Nakamura"
>>> name.upper()
'JORDAN HIROSHI NAKAMURA'
>>> name.lower().startswith("jordan")
True
>>> len(name)
23
```

for Loops and Strings

- A `for` loop can examine each character in a string in order.

```
for name in string:  
    statements
```

```
>>> for c in "booyah":  
...     print(c)  
...  
b  
o  
o  
y  
a  
h
```


while Loops

`while` **test:**
 statements

```
>>> n = 91
>>> factor = 2      # find first factor of n

>>> while n % factor != 0:
...     factor += 1
...

>>> factor
7
```

while / else

```
while test:  
    statements  
  
else:  
    statements
```

- Executes the `else` part if the loop never enters
- There is also a similar `for / else` statement

```
>>> n = 91  
>>> while n % 2 == 1:  
...     n += 1  
... else:  
...     print (n, "was even; no loop.")  
...  
91 was even; no loop.
```



Data Structures

Lists

- **list**: Python's equivalent to Java's array (but cooler)
 - Declaring:
name = [**value**, **value**, ..., **value**] or,
name = [**value**] * **length**
 - Accessing/modifying elements: (same as Java)
name[**index**] = **value**
- Like a string, list is a sequence of values.
- In a string, the values are characters; in a list they can be any type.
- Unlike strings, lists are mutable

```
>>> scores = [9, 14, 12, 19, 16, 18, 24, 15]
>>> fruits = ['apple', 'mango', 'banana']
>>> mixtyp = ['spam', 2.0, 5, [10, 20]]
>>> empty = []
>>> counts = [0] * 4
[0, 0, 0, 0]
>>> scores[0] + scores[4]
25
```

- Nested list counts as a single element.

Indexing

- Lists can be indexed using positive or negative numbers.
- List indices work the same way as string indices

```
>>> scores = [9, 14, 12, 19, 16, 18, 24, 15]
>>> scores[3]
19
>>> scores[-3]
18
```

<i>index</i>	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>
<i>value</i>	9	14	12	19	16	18	24	15
<i>index</i>	<i>-8</i>	<i>-7</i>	<i>-6</i>	<i>-5</i>	<i>-4</i>	<i>-3</i>	<i>-2</i>	<i>-1</i>

Slicing

- **slice:** A sub-list created by specifying start/end indexes

name [**start**:**end**] # **end is exclusive**
name [**start**:] # **to end of list**
name [:**end**] # **from start of list**
name [**start**:**end**:**step**] # **every step'th value**

```
>>> scores = [9, 14, 12, 19, 16, 18, 24, 15]
>>> scores[2:5]
[12, 19, 16]
>>> scores[3:]
[19, 16, 18, 24, 15]
>>> scores[:3]
[9, 14, 12]
>>> scores[-3:]
[18, 24, 15]
```

<i>index</i>	0	1	2	3	4	5	6	7
<i>value</i>	9	14	12	19	16	18	24	15
<i>index</i>	-8	-7	-6	-5	-4	-3	-2	-1

List traversing and operations

- The **in** operator also works on lists.

```
>>> shoplist = ['apple', 'mango', 'banana']  
>>> 'mango' in shoplist  
True  
>>> 'grapes' in shoplist  
False
```

```
>>> shoplist = ['apple', 'mango', 'banana']  
>>> for item in shoplist:  
    print(item)
```

```
>>> a=[1,2,3];b=[4,5,6]  
>>> a+b  
[1,2,3,4,5,6]
```


Other List Abilities

- You can embed a for loop in your array!

```
>>> [i for i in range(20)]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

You can even put an if statement in there!

```
>>> [i for i in range(20) if i % 2 == 0]
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

This can also be accomplished with this:

```
>>> [2*i for i in range(10)]
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Other List Abilities

- Lists can be printed (or converted to string with `str()`).
- Find out a list's length by passing it to the `len` function.
- Loop over the elements of a list using a `for ... in` loop.

```
>>> scores = [9, 14, 18, 19]
>>> print ("My scores are", scores)
My scores are [9, 14, 18, 19]
>>> len(scores)
4
>>> total = 0
>>> for score in scores:
...     print ("next score:", score)
...     total += score
next score: 9
next score: 14
next score: 18
next score: 19
>>> total
60
```

Exercise

- Write a function `prime_numbers` that accepts an `int num` as a parameter and returns an array that has all the primes up to `num`.

Deleting Elements

- Several ways to delete an element from a list

```
>>> t = ['a', 'b', 'c']
>>> x=t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b
>>> del t[1]
>>> print(t)
['c']
>>> t.remove('c')
>>> print(t)
[]
```

- If you don't provide an index, pop deletes and returns the last element.

 The return value from remove is None.

Ranges, Strings, and Lists

- The `range` function returns a list.

```
>>> nums = list(range(5))
>>> nums
[0, 1, 2, 3, 4]
>>> nums[-2:]
[3, 4]
>>> len(nums)
5
```

- Strings behave like lists of characters:

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s', 'p', 'a', 'm']
```

String Splitting

- `split` breaks a string into a list of tokens.

name.split() # break by whitespace

name.split(**delimiter**) # break by delimiter

- `join` performs the opposite of a `split`
delimiter.join(**list**)

```
>>> name = "Brave Sir Robin"
>>> name[-5:]
'Robin'
>>> tokens = name.split()
['Brave', 'Sir', 'Robin']
>>> name.split("r")
['B', 'ave Si', ' Robin']
>>> "||".join(tokens)
'Brave||Sir||Robin'
```

List methods

- Python provides methods that operate on lists.
- Most list methods are void; they modify the list and return None.

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
>>> x = ['e', 'f']
>>> t.extend(x)
>>> print(t)
['a', 'b', 'c', 'd', 'e', 'f']
>>> z = ['b', 'd', 'c']
>>> z.sort()
>>> print(z)
['b', 'c', 'd']
```

Tuple

- A tuple is a sequence of values much like list. Tuples are immutable.

tuple_name = (value, value, ..., value)

- A way of "packing" multiple values into one variable

```
>>> x = 3
>>> y = -5
>>> p = (x, y, 42)
>>> p
(3, -5, 42)
```

name, name, ..., name = tuple_name

- "unpacking" a tuple's contents into multiple variables

```
>>> a, b, c = p
>>> a
3
>>> b
-5
>>> c
42
```


- Most list operators also work on tuples.

```
>>> t1 = ('a',) # a tuple with a single element
>>> t = tuple('lupins')
>>> print(t)
('l', 'u', 'p', 'i', 'n', 's')
```

```
>>> t = ('a', 'b', 'c', 'd')
>>> t[0] = 'A'
TypeError: object does not support item assignment
```

- You can't modify the elements of a tuple, but you can replace one tuple with another

```
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd')
```

Using Tuples

- Useful for storing multi-dimensional data (e.g. (x, y) points)

```
>>> p = (42, 79)
```

- Useful for returning more than one value

```
>>> from random import *
>>> def roll2():
...     die1 = randint(1, 6)
...     die2 = randint(1, 6)
...     return (die1, die2)
...
>>> d1, d2 = roll2()
>>> d1
6
>>> d2
4
```

Tuple as Parameter

```
def name ( (name, name, ..., name) , ... ) :  
    statements
```

- Declares tuple as a parameter by naming each of its pieces

```
>>> def slope((x1, y1), (x2, y2)):  
...     return (y2 - y1) / (x2 - x1)  
...  
>>> p1 = (2, 5)  
>>> p2 = (4, 11)  
>>> slope(p1, p2)  
3
```

Tuple as Return

```
def name (parameters) :  
    statements  
    return (name, name, ..., name)
```

```
>>> from random import *  
>>> def roll2():  
...     die1 = randint(1, 6)  
...     die2 = randint(1, 6)  
...     return (die1, die2)  
...  
>>> d1, d2 = roll2()  
>>> d1  
6  
>>> d2  
4
```

Dictionaries

- A dictionary is like a list, but more general.
- In a list, the index positions have to be integers; in a dictionary, the indices can be (almost) any type.
- You can think of it as a list of pairs, where the first element of the pair, the **key**, is used to retrieve the second element, the **value**.
- Thus we ***map a key to a value***

Key Value Pairs

- The key acts as an index to find the associated value.
- Just like a dictionary, you look up a word by its spelling to find the associated definition
- A dictionary can be searched to locate the value associated with a key
- Key must be immutable
 - strings, integers, tuples are fine
 - lists are NOT
- Value can be anything

Python Dictionary

- Use the { } marker to create a dictionary
- Use the : marker to indicate key:value pairs

```
>>> contacts= {'bill': '353-1234', 'rich': '269-1234',  
'jane': '352-1234'}  
>>> print (contacts)  
{'jane': '352-1234', 'bill': '353-1234', 'rich': '369-  
1234'}
```

- One can also use the function `dict()` to create a new dictionary with no items.

Collections but not a Sequence

- Dictionaries are collections but they are not sequences such as lists, strings or tuples
 - there is no order to the elements of a dictionary
 - in fact, the order (for example, when printed) might change as elements are added or deleted.
- So how to access dictionary elements?

Access Dictionary Elements

The *key* is the index!

```
my_dict={}
```

- an empty dictionary

```
my_dict['bill']=25
```

- added the pair 'bill':25

```
print(my_dict['bill'])
```

- prints 25

Dictionaries are mutable

- Like lists, dictionaries are a mutable data structure
 - you can change the object via various operations, such as index assignment

```
my_dict = {'bill':3, 'rich':10}
print(my_dict['bill'])      # prints 3
my_dict['bill'] = 100
print(my_dict['bill'])      # prints 100
```

Common operators

Like others, dictionaries respond to these

- `len(my_dict)`
 - number of key:value **pairs** in the dictionary
- `element in my_dict`
 - boolean, is `element` a **key** in the dictionary
- `for key in my_dict:`
 - iterates through the **keys** of a dictionary

Dictionary Methods

Only 9 methods in total. Here are some

- `key in my_dict`
does the key exist in the dictionary
- `my_dict.clear()` – empty the dictionary
- `my_dict.update(yourDict)` – for each key in `yourDict`, **updates** `my_dict` with that key/value pair
- `my_dict.copy` - shallow copy
- `my_dict.pop(key)` – remove key, return value

Dictionary content methods

- `my_dict.items()` – all the key/value pairs
- `my_dict.keys()` – all the keys
- `my_dict.values()` – all the values

They return what is called a *dictionary view*.

- the order of the views correspond
- are dynamically updated with changes
- are iterable

Views are iterable

```
for key in my_dict:
```

```
    print(key)
```

- prints all the keys

```
for key,value in my_dict.items():
```

```
    print (key,value)
```

- prints all the key/value pairs

```
for value in my_dict.values():
```

```
    print (value)
```

- prints all the values

```
• my_dict = {'a':2, 3:['x', 'y'], 'joe':'smith'}
```

```
>>> dict_value_view = my_dict.values()
```

```
>>> dict_value_view # a view
```

```
dict_values([2, ['x', 'y'], 'smith'])
```

```
>>> type(dict_value_view) # view type
```

```
<class 'dict_values'>
```

```
>>> for val in dict_value_view: # view iteration  
    print(val)
```

```
2
```

```
['x', 'y']
```

```
smith
```

```
>>> my_dict['new_key'] = 'new_value'
```

```
>>> dict_value_view # view updated
```

```
dict_values([2, 'new_value', ['x', 'y'], 'smith'])
```

```
>>> dict_key_view = my_dict.keys()
```

```
dict_keys(['a', 'new_key', 3, 'joe'])
```

```
>>> dict_value_view
```

```
dict_values([2, 'new_value', ['x', 'y'], 'smith']) # same order
```

Example

- Suppose you are given a string and you want to count how many times each letter appears

```
words='brontosaurus'  
d = dict()  
for c in words:  
    if c not in d:  
        d[c]=1  
    else:  
        d[c]=d[c]+1  
print(d)
```


Another way

```
words='brontosaurus'
d = dict()
for c in words:
    d[c]=d.get(c,0)+1
print(d)

#  get returns the corresponding value;
#  otherwise it returns the default value.
```

Sets

- in mathematics, a set is a collection of objects, potentially of many different types
- in a set, no two elements are identical. That is, a set consists of elements each of which is unique compared to the other elements
- there is no order to the elements of a set
- a set with no elements is the empty set

Creating a Set

Set can be created in one of two ways:

- `set(iterable)` where the argument is iterable

```
my_set = set('abc')
```

```
my_set → {'a', 'b', 'c'}
```

- shortcut: `{}`, braces where the elements have no colons (to distinguish them from dictionary)

```
my_set = {'a', 'b', 'c'}
```

Diverse Elements; no duplicates

- A set can consist of a mixture of different types of elements

```
my_set = {'a', 1, 3.14159, False}
```

- duplicates are automatically removed

```
my_set = set("aabbccdd")
```

```
print(my_set)
```

```
→ {'a', 'c', 'b', 'd'}
```

Examples

```
>>> null_set = set()           # set() creates the empty set
>>> null_set
set()
>>> a_set = {1,2,3,4}          # no colons means set
>>> a_set
{1, 2, 3, 4}
>>> b_set = {1,1,2,2,2}        # duplicates are ignored
>>> b_set
{1, 2}
>>> c_set = {'a', 1, 2.5, (5,6)} # different types is OK
>>> c_set
{(5, 6), 1, 2.5, 'a'}
>>> a_set = set("abcd")        # set constructed from iterable
>>> a_set
{'a', 'c', 'b', 'd'}          # order not maintained!
```

Common Operators

Most data structures respond to these:

- `len(my_set)`
 - the number of elements in a set
- `element in my_set`
 - boolean indicating whether element is in the set
- `for element in my_set:`
 - iterate through the elements in `my_set`

Set Operators

- The set data structure provides some special operators that correspond to the operators you learned in middle school.
- These are various combinations of set contents.
- These operations have both a method name and a shortcut binary operator.

method: intersection, op: &

```
a_set=set("abcd") b_set=set("cdef")
```

- `a_set & b_set` → `{'c', 'd'}`
- `b_set.intersection(a_set)` → `{'c', 'd'}`

Difference; Union

method: difference op: -

```
a_set=set("abcd") b_set=set("cdef")
```

- `a_set - b_set` → `{'a', 'b'}`
- `b_set.difference(a_set)` → `{'e', 'f'}`

method: union, op: |

- `a_set | b_set` → `{'a', 'b', 'c', 'd', 'e', 'f'}`
- `b_set.union(a_set)` → `{'a', 'b', 'c', 'd', 'e', 'f'}`

Symmetric Difference

method: `symmetric_difference`, op: `^`

- `a_set ^ b_set` \rightarrow `{'a', 'b', 'e', 'f'}`
- `b_set.symmetric_difference(a_set)` \rightarrow `{'a', 'b', 'e', 'f'}`

method: `issubset`, op: `<=`

method: `issuperset`, op: `>=`

`small_set=set("abc"); big_set=set("abcdef")`

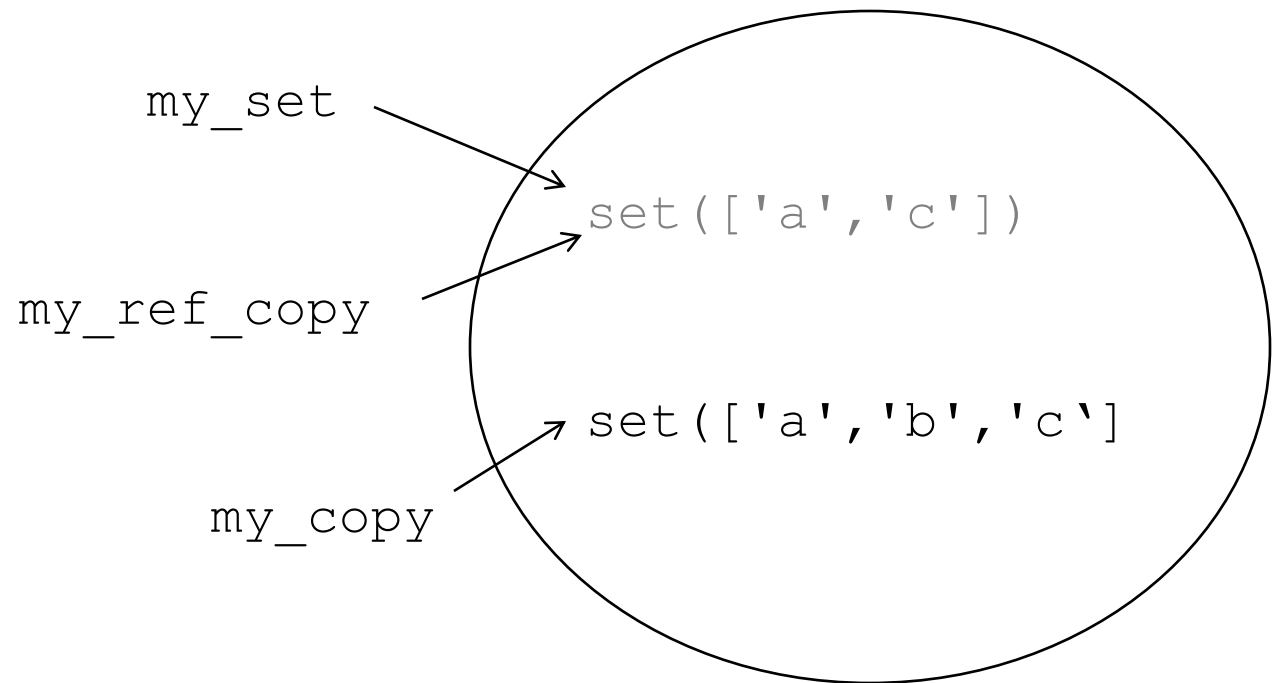
- `small_set <= big_set` \rightarrow `True`
- `big_set >= small_set` \rightarrow `True`

Other Set Ops

- `my_set.add("g")`
 - adds to the set, no effect if item is in set already
- `my_set.clear()`
 - empties the set
- `my_set.remove("g")` **versus**
`my_set.discard("g")`
 - `remove` throws an error if "g" isn't there. `discard` doesn't care. Both remove "g" from the set
- `my_set.copy()`
 - returns a shallow copy of `my_set`

Copy vs assignment

- `my_set={'a', 'b', 'c'}`
- `my_copy=my_set.copy()`
- `my_ref_copy=my_set`
- `my_set.remove('b')`



Shallow and deep copy operations

- Assignment statements in Python do not copy objects, they create bindings between a target and an object.
- The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):
- A *shallow copy* constructs a new compound object and then (to the extent possible) inserts *references* into it to the objects found in the original. **(`copy()`)**
- A *deep copy* constructs a new compound object and then, recursively, inserts *copies* into it of the objects found in the original. **(`copy.deepcopy()`)** **#first import copy module**

```
>>> xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> ys = xs.copy() # Make a shallow copy
>>> xs.append(['new sublist'])
>>> xs
[[1, 2, 3], [4, 5, 6], [7, 8, 9], ['new sublist']]
>>> ys
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

- However, because we only created a *shallow* copy of the original list, `ys` still contains references to the original child objects stored in `xs`.
- Therefore, when you modify one of the child objects in `xs`, this modification will be reflected in `ys` as well—that's because *both lists share the same child objects*. The copy is only a shallow, one level deep copy:

```
>>> xs[1][0] = 'X'
>>> xs
[[1, 2, 3], ['X', 5, 6], [7, 8, 9], ['new sublist']]
>>> ys
[[1, 2, 3], ['X', 5, 6], [7, 8, 9]]
```

- Had we created a *deep* copy of `xs` in the first step, both objects would've been fully independent.

```
>>> import copy
>>> xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> zs = copy.deepcopy(xs)
```



File Processing

What is a file?

- A file is a collection of data that is stored on secondary storage like a disk or a thumb drive
- accessing a file means establishing a connection between the file and the program and moving data between the two

File Objects or stream

- When opening a file, you create a file object or file stream that is a connection between the file information on disk and the program.
- The stream contains a buffer of the information from the file, and provides the information to the program

Reading Files

name = open(filename, mode) # nm=open('hours.txt', 'r')

– opens the given file for reading, and returns a file object

name.read() – file's entire content as a string

- filename is a string
- mode is optional and should be 'r' if we are planning to read the file and 'w' if we are going to write to the file
- The open function creates the connection between the disk file and the file object.

```
>>> f = open("mbox-short.txt")
>>> inp=f.read()
>>> print(inp)
```

Different modes

Mode	How Opened	File Exists	File Does Not Exist
'r'	read-only	Opens that file	Error
'w'	write-only	Clears the file contents	Creates and opens a new file
'a'	write-only	File contents left intact and new data appended at file's end	Creates and opens a new file
'r+'	read and write	Reads and overwrites from the file's beginning	Error
'w+'	read and write	Clears the file contents	Creates and opens a new file
'a+'	read and write	File contents left intact and read and write at file's end	Creates and opens a new file

 File Modes

Line-based File Processing

- name.readline()** – next line from file as a string
– Returns an empty string if there are no more lines in the file
- name.readlines()** – file's contents as a list of lines

```
>>> f = open("mbox-short.txt")
>>> inp = f.readline()
>>> print(inp)

>>> f = open("mbox-short.txt")
>>> inp = f.readlines()
```

File Handle as a Sequence

- A file handle open for read can be treated as a sequence of strings where each line in the file is a string in the sequence
- We can use the for statement to iterate through a sequence

```
for line in open("filename") :  
    statements
```

```
>>> count = 0  
...for line in open("filename") :  
.....     count = count + 1  
...print('Line Count:', count)
```

Searching Through a File

- We can put an if statement in our for loop to only print lines that meet some criteria

```
fhand = open('mbox-short.txt')
for line in fhand:
    if line.startswith('From:') :
        print(line)
```

- Observe the blank lines in the output. Why?
 - Each line from the file has a newline at the end
 - The print statement adds a newline to each line

Searching Through a File (fixed)

- We can strip the whitespace from the right-hand side of the string using `rstrip()` from the string library
- The newline is considered “white space” and is stripped

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.startswith('From:') :
        print(line)
```

Skipping with continue

We can conveniently skip a line by using the continue statement

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if not line.startswith('From: ') :
        continue
    print(line)
```

Writing Files

```
name = open ("filename", "w")      # write  
name = open ("filename", "a")      # append
```

- opens file for write (deletes any previous contents) , or
- opens file for append (new data is placed after previous data)

name.write(str) – writes the given string to the file

name.close() – closes file once writing is done

```
>>> out = open("output.txt", "w")  
>>> out.write("Hello, world!\n")  
>>> out.write("How are you?")  
>>> out.close()  
  
>>> open("output.txt").read()  
'Hello, world!\nHow are you?'
```


Exercise

- Write a function `stats` that accepts a file name as a parameter and that reports the longest line in the file.
 - example input file, `vendetta.txt`:

Remember, remember the 5th of November.

The gunpowder, treason, and plot.

I know of no reason why the gunpowder treason
should ever be forgot.

- expected output:

```
>>> stats("vendetta.txt")
longest line = 46 characters
I know of no reason why the gunpowder treason
```

Exercise Solution

```
def stats(filename):  
    longest = ""  
    for line in open(filename):  
        if len(line) > len(longest):  
            longest = line  
  
    print("Longest line =", len(longest))  
    print(longest)
```

Exercise

- Write a function `remove_lowercase` that accepts two file names and copies the first file's contents into the second file, with any lines that start with lowercase letters removed.

- example input file, `carroll.txt`:

```
Beware the Jabberwock, my son,  
the jaws that bite, the claws that catch,  
Beware the JubJub bird and shun  
the frumious bandersnatch.
```

- expected behavior:

```
>>> remove_lowercase("carroll.txt", "out.txt")  
>>> print(open("out.txt").read())  
Beware the Jabberwock, my son,  
Beware the JubJub bird and shun
```

Exercise Solution

```
def remove_lowercase(infile, outfile):  
    output = open(outfile, "w")  
    for line in open(infile):  
        if line[0].isupper():  
            output.write(line)  
    output.close()
```



Exceptions

Excetions

- Most modern languages provide methods to deal with 'exceptional' situations
- this is not about fundamental CS, but about doing a better job as a programmer

What counts as exceptional

- Errors: indexing past the end of a list, trying to open a nonexistent file, fetching a nonexistent key from a dictionary, etc
- Ending conditions: File should be closed at the end of processing

Error Names

- Errors have specific names, and Python shows them to us all the time.

```
>>> input_file = open("no_such_file.txt", 'r')
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    input_file = open("no_such_file.txt", 'r')
IOError: [Errno 2] No such file or directory: 'no_such_file.txt'
>>> my_int = int('a string')
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    my_int = int('a string')
ValueError: invalid literal for int() with base 10: 'a string'
>>>
```

You can recreate an error to find the correct name. Spelling

A kind of non-local control

Basic Idea

- keep watch on a particular section of code
- if we get an exception, raise/throw that exception (let it be known)
- look for a catcher that can handle that kind of exception
- if found, handle it, otherwise let Python handle it (which usually halts the program)

General Form

```
try:  
    suite  
except a_particular_error:  
    suite
```

try suite

- the `try` suite contains code that we want to monitor for errors during its execution.
- if an error occurs anywhere in that `try` suite, Python looks for a handler that can deal with the error.
- if no special handler exists, Python handles it, meaning the program halts and with an error message as we have seen so many times ☹️

except suite

- an `except` suite (perhaps multiple `except` suites) is associated with a `try` suite.
- each exception names a type of exception it is monitoring for.
- if the error that occurs in the `try` suite matches the type of exception, then that `except` suite is activated.

try/except group

- if no exception in the `try` suite, skip all the `try/except` to the next line of code
- if an error occurs in a `try` suite, look for the right exception
- if found, run that `except` suite and then skip past the `try/except` group to the next line of code
- if no exception handling found, give the error to Python

Bad File Names

```
fname = input('Enter the file name:  ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    quit()
count = 0
for line in fhand:
    if line.startswith('Subject:') :
        count = count + 1
print('There were', count, 'subject lines in',
      fname)
```



Classes

What is a class?

- You likely will have heard the term object oriented programming (OOP)
- What is OOP, and why should I care?

Short answer

- The short answer is that object oriented programming is a way to think about “objects” in a program (such as variables, functions, etc)
- A program becomes less a list of instruction and more a set of objects and how they interact

OOP principle

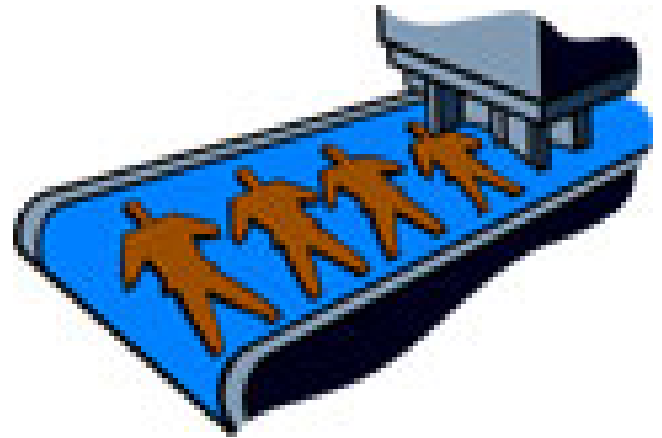
- ***encapsulation***: hiding design details to make the program clearer and more easily modified later
- ***abstraction***: to deal with objects considering their important characteristics and ignore all other details.
- ***inheritance***: create a new object by inheriting (like father to son) many object characteristics while creating or over-riding for this object
- ***polymorphism***: (hard) Allow one message to be sent to any object and have it respond appropriately based on the type of object it is.

OOP and Python

- OOP exists and works fine, but feels a bit more "tacked on"
- Java probably does classes better than Python (gasp)

Class versus instance

- One of the harder things to get is what a class is and what an instance of a class is.
- The analogy of the cookie cutter and a cookie.



Template vs exemplar

- The cutter is a template for stamping out cookies, the cookie is what is made each time the cutter is used
- One template can be used to make an infinite number of cookies, each one just like the other.
- No one confuses a cookie for a cookie cutter, do they?

Same in OOP

- You define a class as a way to generate new instances of that class.
- Objects are nothing but instance of the class.
- the structure of an instance starts out the same, as dictated by the class.
- The instances respond to the messages defined as part of the class.

Why a class

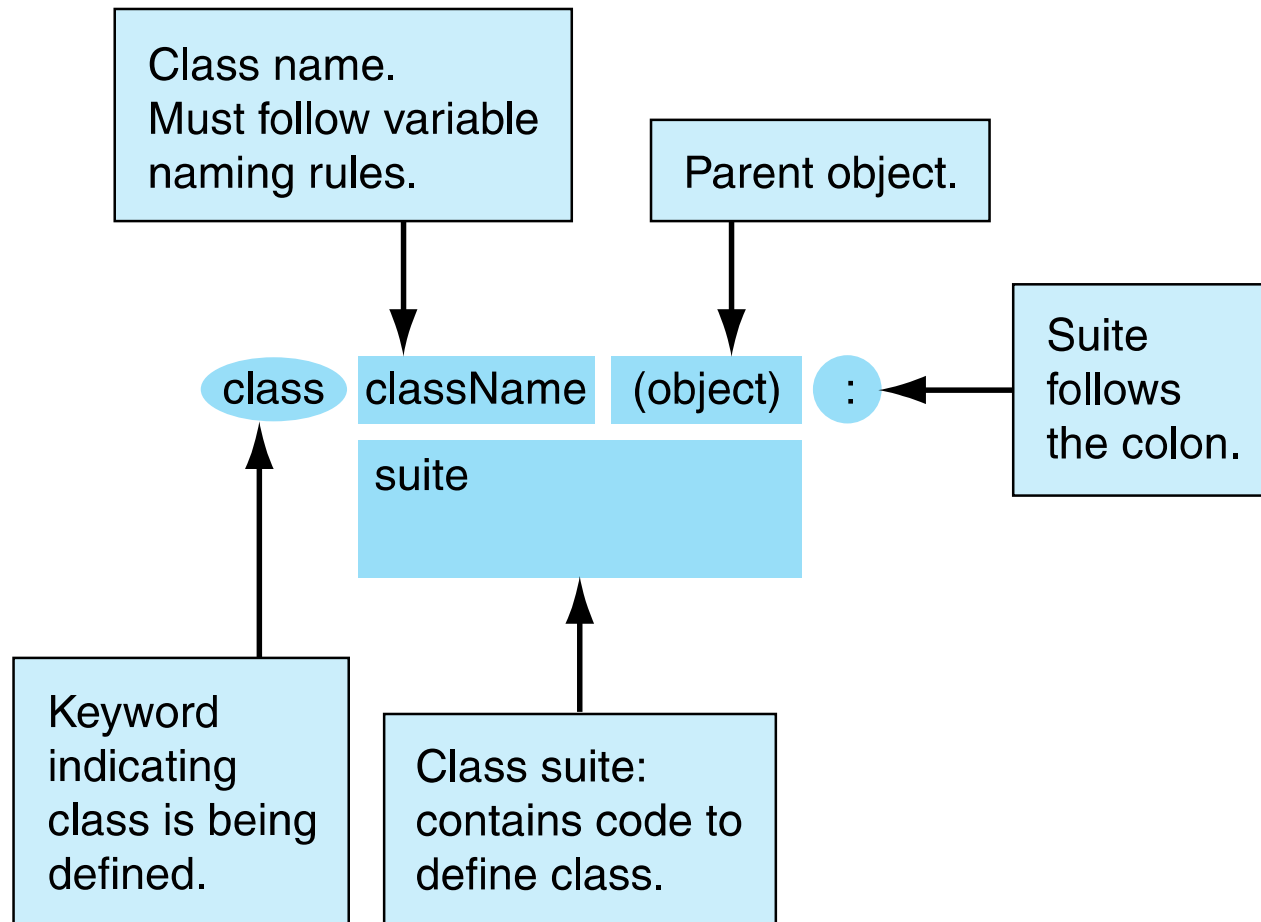
- Classes allow us to logically group out attributes and methods in a way that is easy to reuse and way to build upon if need be.
- Each class has potentially two aspects:
 - the data (types, number, names) that each instance might contain
 - the messages that each instance can respond to.

Standard Class Names

The standard way to name a class in Python is called ***CapWords*** :

- Each word of a class begins with a Capital letter
- no underlines
- makes recognizing a class easier

Defining a Class



The basic format of a class definition.

dir() function and pass keyword

- The `dir()` function lists all the attributes of a class

The `pass` keyword is used to signify that you have *intentionally* left some part of a definition (of a function, of a class) undefined

- by making the suite of a class undefined, we get only those things that Python defines for us automatically


```
>>> class MyClass (object):  
    pass
```

```
>>> dir(MyClass)  
'__class__', '__delattr__', '__dict__', '__doc__', '__eq__',  
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',  
'__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__',  
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',  
'__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

```
>>> my_instance = MyClass()  
>>> dir(my_instance)  
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__',  
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',  
'__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__',  
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',  
'__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

```
>>> type(my_instance)  
<class '__main__.MyClass'>
```

dot reference

- we can refer to the attributes of an object by doing a dot reference, of the form:
`object.attribute`
- the attribute can be a variable or a function
- it is part of the object, either directly or by that object being part of a class

Examples:

```
print(my_instance.my_val)
```

print a variable associated with the object `my_instance`

```
my_instance.my_method()
```

call a method associated with the object `my_instance`

How to make an object-local value

- once an object is made, the data is made the same way as in any other Python situation, by assignment
- Any object can thus be augmented by adding a variable

```
my_instance.attribute = 'hello'
```

- New attributes shown in dir

```
dir(my_instance)
```

```
- ['__class__', '__delattr__', '__dict__',  
  '__doc__', '__format__', '__getattr__',  
  '__hash__', '__init__', '__module__', '__new__',  
  '__reduce__', '__reduce_ex__', '__repr__',  
  '__setattr__', '__sizeof__', '__str__',  
  '__subclasshook__', '__weakref__', 'attribute']
```

Instance knows its class

- Because each instance has as its type the class that it was made from, an instance remembers its class
- This is often called the ***instance-of*** relationship
- stored in the `__class__` attribute of the instance

```
>>> class MyClass (object):  
    pass
```

```
>>> my_instance = MyClass()  
>>> MyClass.class_attribute = 'hello'  
>>> my_instance.instance_attribute = 'world'  
>>> dir(my_instance)  
['__class__', ... , 'class_attribute', 'instance_attribute']  
>>> print(my_instance.__class__)  
<class '__main__.MyClass'>  
  
>>> type(my_instance)  
<class '__main__.MyClass'>  
>>> print(my_instance.instance_attribute)  
world  
>>> print(my_instance.class_attribute)  
hello  
>>> print(MyClass.instance_attribute)
```

```
Traceback (most recent call last):
```

```
File "<pyshell#11>", line 1, in <module>
```

```
    print MyClass.instance_attribute
```

```
AttributeError: type object 'MyClass' has  
no attribute 'instance_attribute'
```

Scope

Part of the Object Scope Rule

The first two rules in object scope are:

1. First, look in the object itself
2. If the attribute is not found, look up to the class of the object and search for the attribute there.

```
>>> class MyClass (object):  
        pass
```

```
>>> inst1 = MyClass()  
>>> inst2 = MyClass()  
>>> inst3 = MyClass()  
>>> MyClass.class_attribute = 27  
>>> inst1.class_attribute = 72  
>>> print(inst1.class_attribute)  
72  
>>> print(inst2.class_attribute)  
27  
>>> print(inst3.class_attribute)  
27  
>>> MyClass.class_attribute = 999  
>>> print(inst1.class_attribute)  
72  
>>> print(inst2.class_attribute)  
999  
>>> print(inst3.class_attribute)  
999
```

Using Simple Classes

```
class Point:  
    pass  
p = Point()  
p.x=2.0  
p.y=3.3
```


Classes - constructor

```
class Point:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
p = Point(2.0, 3.0)  
print(p.x, p.y)
```

- If you don't provide a constructor, then only the default constructor is provided
- The default constructor does system stuff to create the instance, nothing more
- Every class should have `__init__`
- By providing the constructor, we ensure that every instance, at least at the point of construction, is created with the same contents
- This gives us some control over each instance.

Methods on Objects

```
import math
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def norm(self):
        n=math.sqrt(self.x**2+self.y**2)
        return n
p = Point(2.0, 3.0)
print(p.x, p.y, p.norm())
```

- methods always bind the first parameter in the definition to the object that called it
- This parameter can be named anything, but traditionally it is named ***self***

More on self

- `self` is an important variable. In any method it is bound to the object that called the method
- through `self` we can access the instance that called the method (and all of its attributes as a result)
- when a dot method call is made, the object that called the method is **automatically** assigned to `self`

__str__, printing

```
>>> p = Point(2.0, 3.0)
>>> print (p)
<__main__.Point instance at 0x402da8c
```

- When `print(p)` called, it is assumed, by Python, to be a call to “convert the instance to a string”, which is the `__str__` method
- In the method, `p` is bound to `self`, and printing then occurs using that instance.
- `__str__` ***must return a string!***
- The behavior can be overwritten by defining the following method

```
def __str__(self):
    return "({},{})".format(self.x, self.y)
```

```
>>> p = Point(2.0, 3.0)
>>> print (p)
(2.0,3.0)
```

Comparing objects

```
>>> p1 = Point(2.0, 3.0)
>>> p2 = Point(2.0, 3.0)
>>> p1 == p2
False
```

- The behavior can be overwritten.

```
def __eq__(self, other):
    return (self.x == other.x) and (self.y == other.y)
```

```
>>> p1 == p2    # Check for equal values
True
>>> p1 is p2    # Check for identity
False
```

how does `p1 == p2` map to `__eq__`

`p1 == p2`

is turned, by Python, into

`p1.__eq__(p2)`

- These are ***exactly equivalent expressions***. It means that the first variable calls the `__eq__` method with the second variable passed as an argument
- `p1` is bound to `self`, `p2` bound to `other`

Operator overloading

More relational operators:

- `<` : `__lt__(self, other)`
- `<=` : `__le__(self, other)`
- `!=` : `__ne__(self, other)`
- `>` : `__gt__(self, other)`
- `>=` : `__ge__(self, other)`

• Numeric operators:

- `+` : `__add__(self, other)`
- `-` : `__sub__(self, other)`
- `*` : `__mul__(self, other)`

Class variables

```
class Point:
    count=0    # count all point object
    def __init__(self, x, y):
        Point.count += 1 # self.__class__.count +=1
p1=Point(2,3)
p2=Point(3,4)
print(p1.count, p2.count, Point.count)
```

```
>>> 2 2 2
```


Inheritance

There are often classes that are very similar to each other.

Inheritance allows for:

- Hierarchical class structure (is-a-relationship)
- Reusing of similar code

Example: Different types of phones

- Phone
- Mobile phone (is a phone with additional functionality)
- Smart phone (is a mobile phone with additional functionality)

class statement

name of the class above
this class in the hierarchy



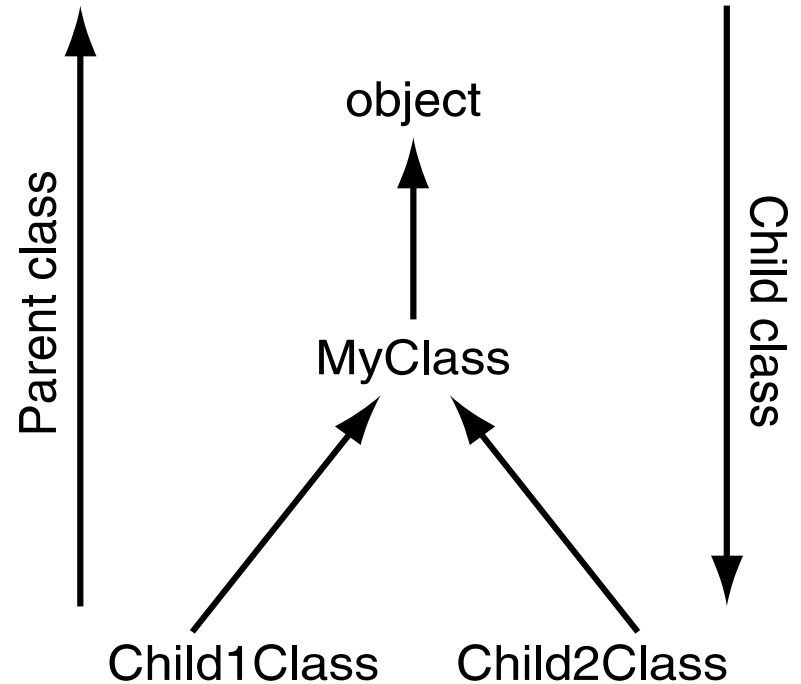
```
class MyClass (SuperClass) :  
    pass
```

- The top class in Python is called `object`.
- it is predefined by Python, always exists
- use `object` when you have no superclass

```
class MyClass (object):  
    pass
```

```
class Child1Class (MyClass):  
    pass
```

```
class Child2Class (MyClass):  
    pass
```



A simple class hierarchy.

Scope

- Look in the object for the attribute
- If not in the object, look to the object's class for the attribute
- If not in the object's class, look up the hierarchy of that class for the attribute

Example

```
class Phone :  
    def call ( self ):  
        pass
```

```
class MobilePhone ( Phone ):  
    def send_text ( self ):  
        pass
```

MobilePhone now inherits methods and attributes from Phone

```
h = MobilePhone ()  
h. call ()      # inherited from Phone  
h. send_text () # own method
```

Methods of the parent class can be overwritten in the child class:

```
class MobilePhone ( Phone ):  
    def call ( self ):  
        find_signal ()  
        Phone . call ( self )
```

Multiple Inheritance

Classes can inherit from multiple parent classes. Example:

- SmartPhone is a mobile phone
- SmartPhone is a camera

```
class SmartPhone ( MobilePhone, Camera):  
    pass
```

```
h = SmartPhone()  
h.call()    #inherited from MobilePhone  
h.take_photo() #inherited from Camera
```

Attributes are searched for in the following order:

SmartPhone , MobilePhone , parent class of MobilePhone (recursively),
Camera, parent class of Camera (recursively).

Private Attributes / Private Class Variables

- private means not accessible by the class user, only the class developer.
- There are no private variables or private methods in Python.
- **Convention:** Mark attributes that shouldn't be accessed from outside with an underscore: `_foo`.
- To avoid name conflicts during inheritance: Names of the form `__foo` are replaced with `_classname__foo`:

```
class Spam:
```

```
    __eggs = 3
```

```
    _bacon = 2
```

```
    beans = 5
```

```
>>> dir(Spam)
```

```
['_Spam__eggs', '__doc__', '__module__', '_bacon', 'beans']
```

Privacy example

```
class NewClass (object):
    def __init__(self, attribute='default', name='Instance'):
        self.name = name                # public attribute
        self.__attribute = attribute    # a "private" attribute
    def __str__(self):
        return '{} has attribute {}'.format(self.name, self.__attribute)
```

```
>>> inst1 = NewClass(name='Monty', attribute='Python')
>>> print(inst1)
Monty has attribute Python
>>> print(inst1.name)
Monty
>>> print(inst1.__attribute)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print(inst1.__attribute)
AttributeError: 'newClass' object has no attribute '__attribute'
>>> dir(inst1)
['_NewClass__attribute', '__class__', ... , 'name']

>>> print(inst1._NewClass__attribute)
Python
```