MIT
Global Startup Labs
México 2013

http://aiti.mit.edu

Lesson 2- Intermediate Python

MISTI

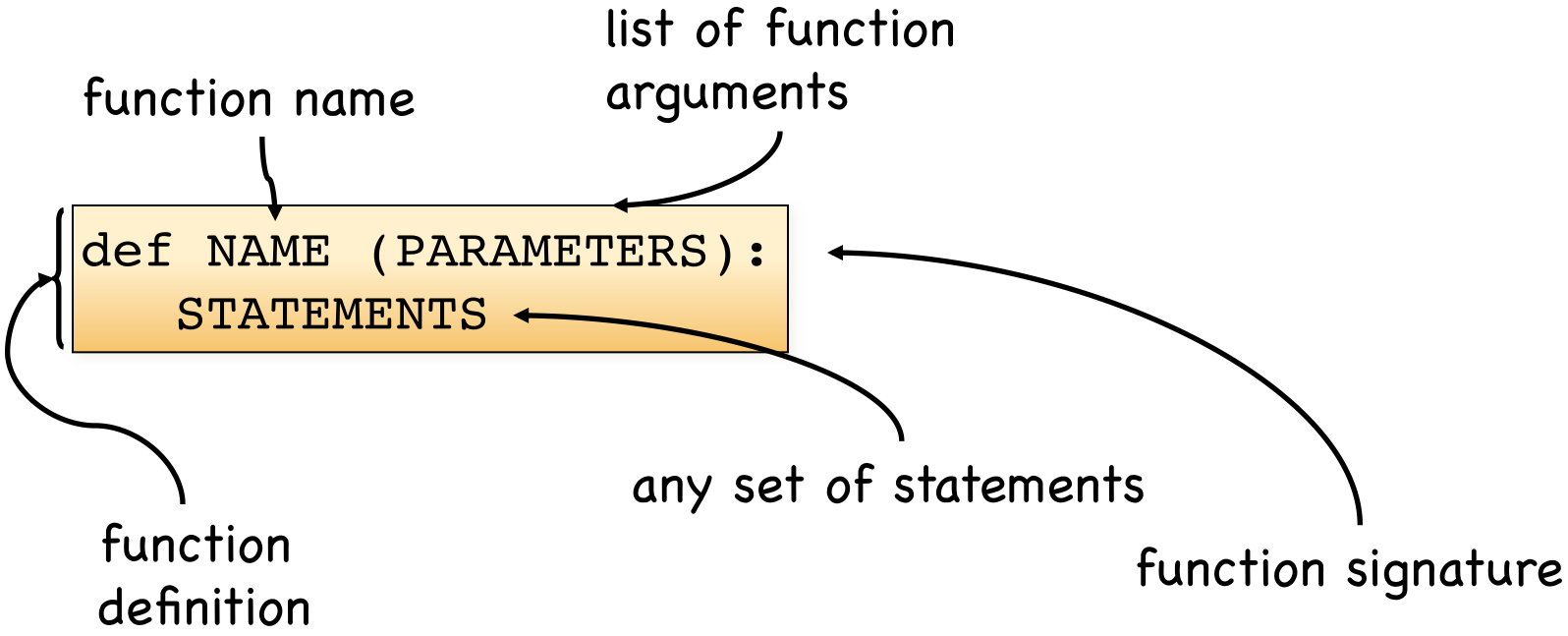
Google

Agenda

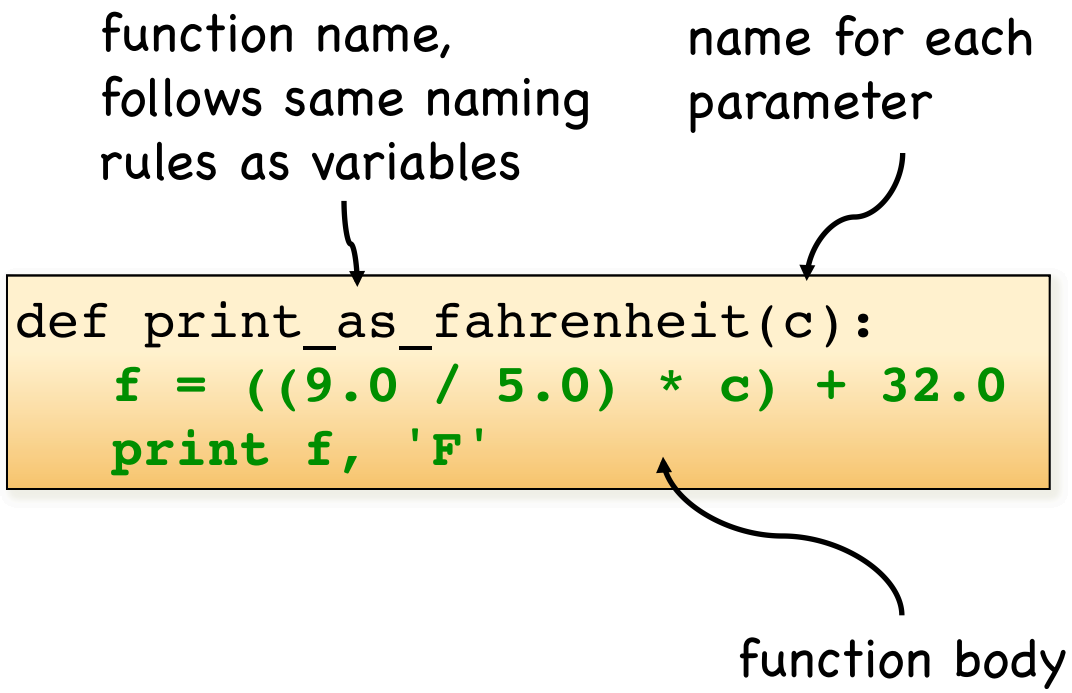
- More on Functions
- Objects
- Exceptions
- Regular Expressions

Functions

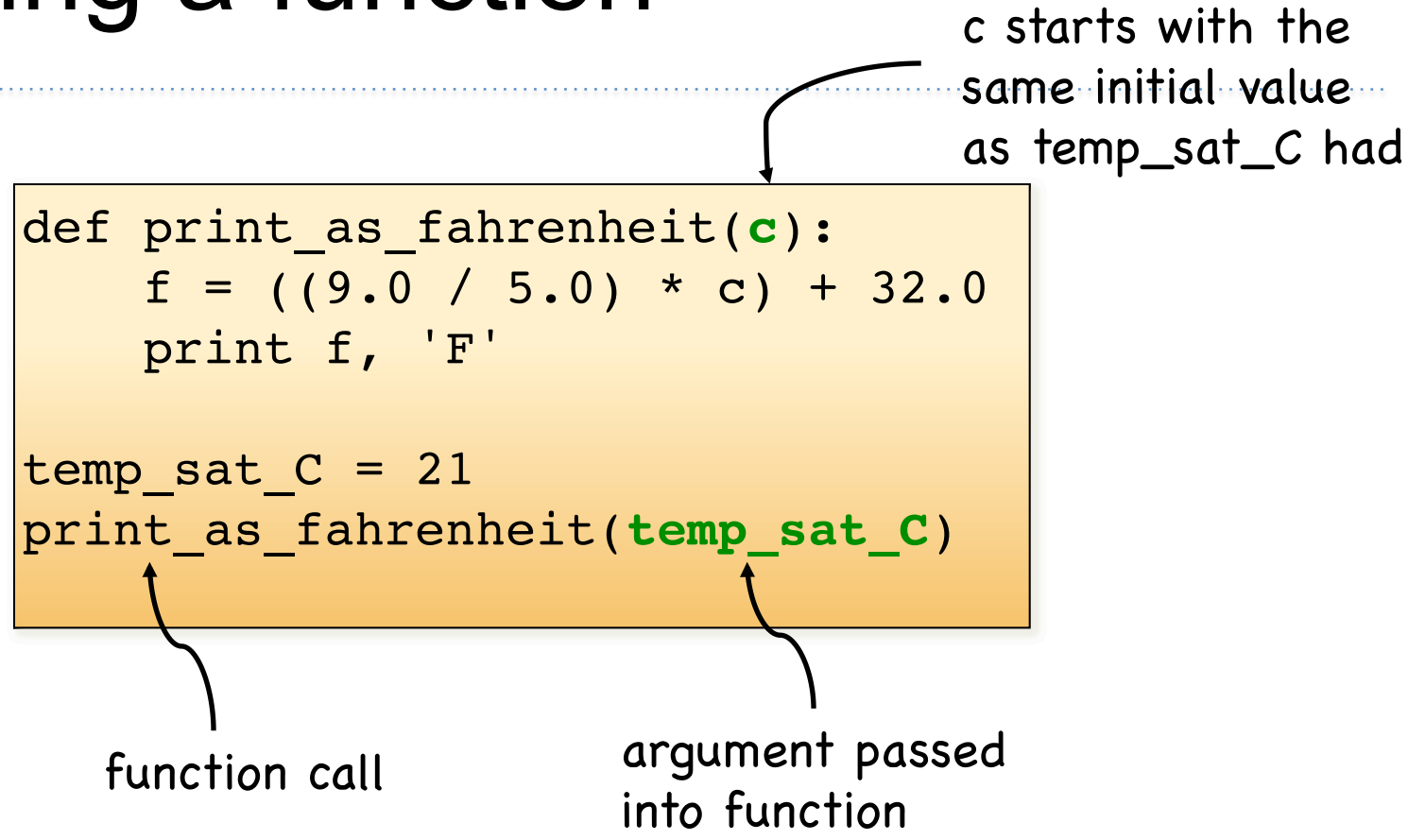
- A function is a sequence of statements that has been given a name.



Defining a function

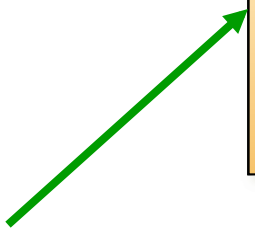


Calling a function



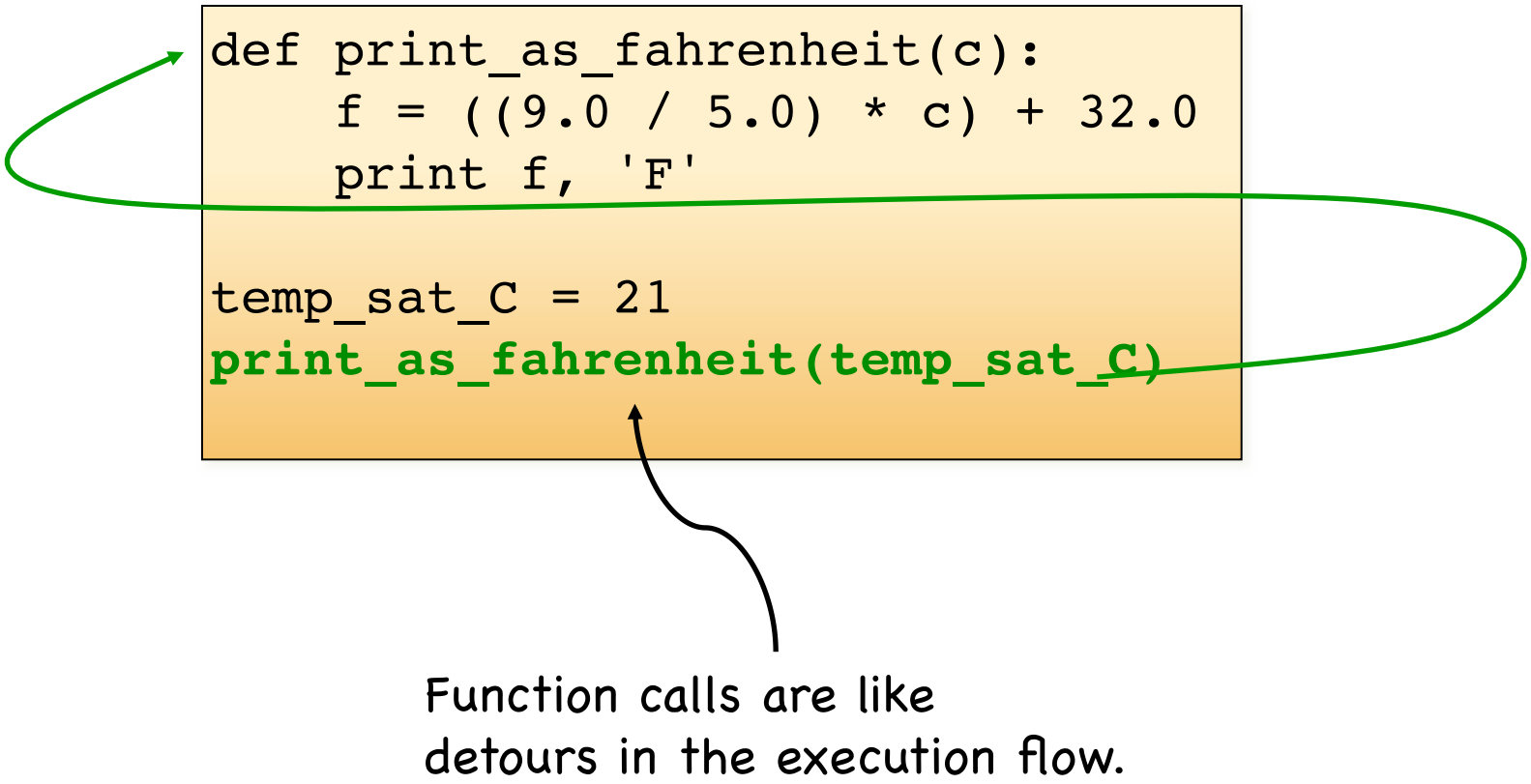
Flow of execution

```
def print_as_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



Program execution always starts at the first line that is *not* a statement inside a function

Flow of execution



Flow of execution

```
def print_as_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

Flow of execution

```
def print_as_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

```
69.80000000000001 F
```

Flow of execution

```
def print_as_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

```
69.80000000000001 F
```

Returning a value

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f
```

return statement

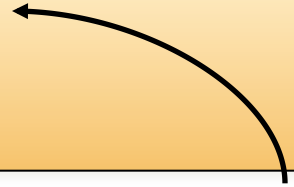
```
return EXPRESSION
```

A return statement ends the function immediately.

any expression, or nothing

More than one return statement

```
def absolute_value(c):
    if c < 0:
        return -c
    else:
        return c
```



If c is negative, the function returns here.

More than one return statement

```
def absolute_value(c):
    if c < 0:
        return –c
    return c
```

Good rule: Every path through the function must have a return statement.
If you don't add one, Python will add one for you that returns nothing (the value None).

What is wrong here?

this function has to be defined before it is called

```
temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f
```

```
NameError: name
'print_as_fahrenheit'
is not defined
```

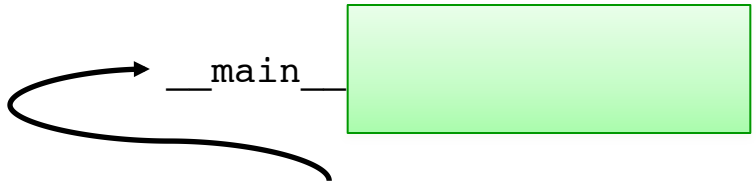
what about this one?

The two functions are in the same level. Therefore, one function can call the other functions even if it is defined after the calling function.

Scoping in functions

- A stack diagram keeps track of where each variable is defined, and its value.

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

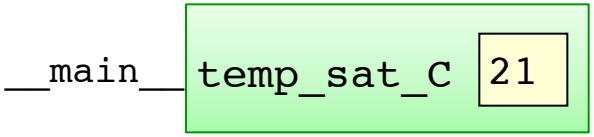


Each function call gets its own stack frame.

Scoping in functions

- A stack diagram keeps track of where each variable is defined, and its value.

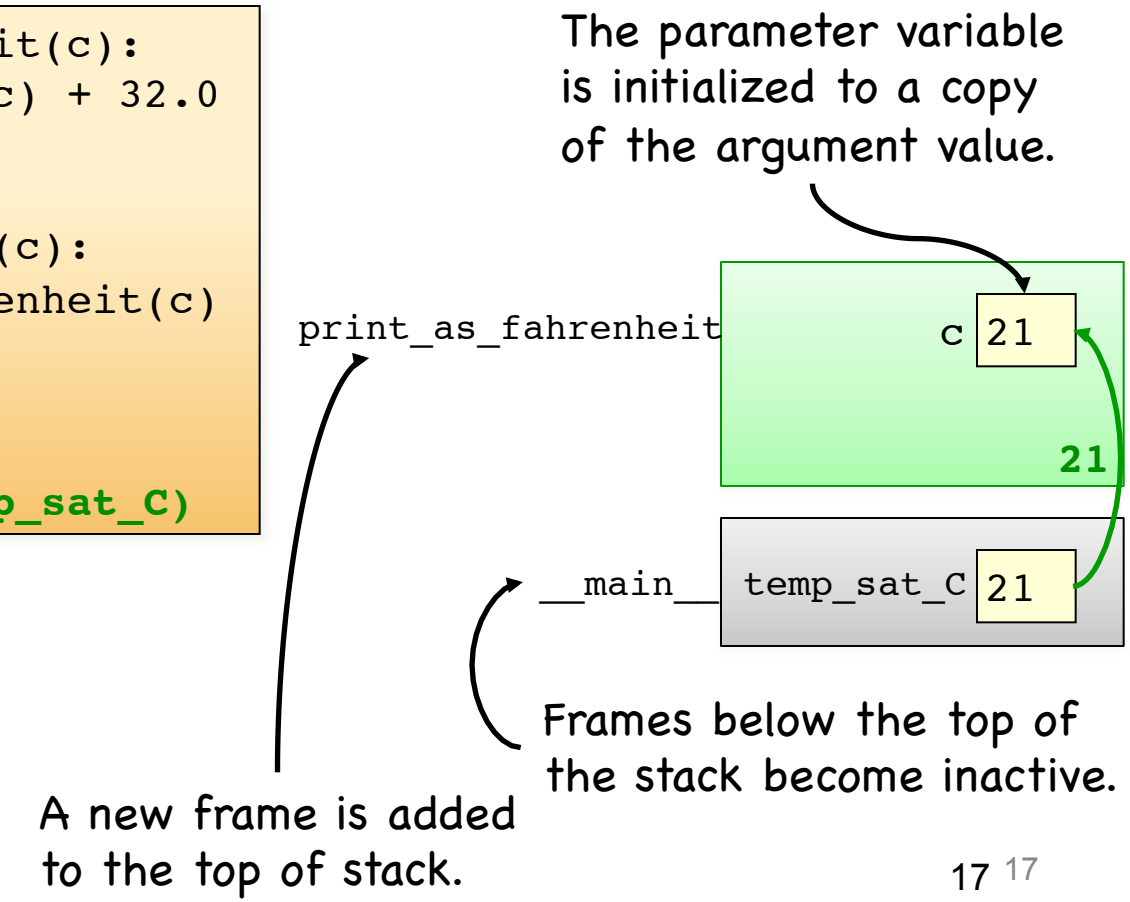
```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



Scoping in functions

- A stack diagram keeps track of where each variable is defined, and its value.

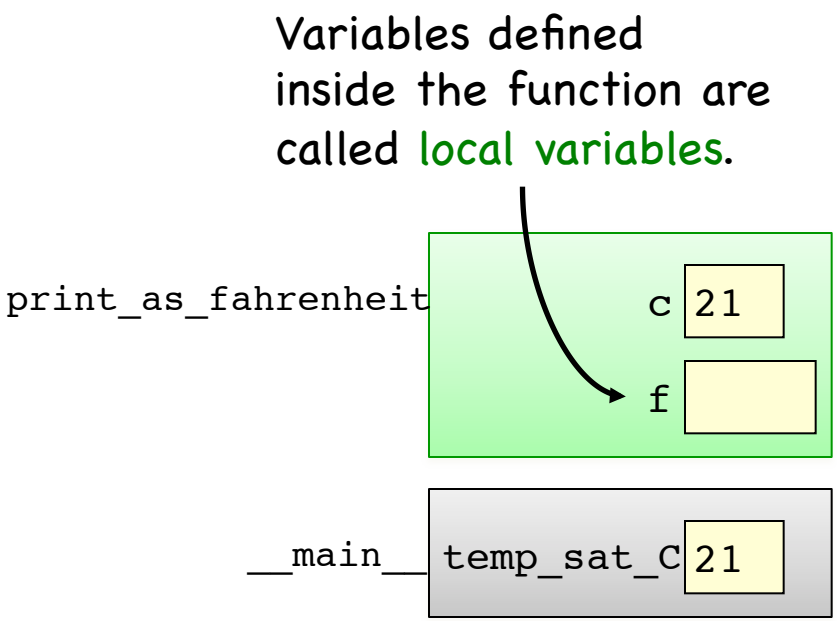
```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



Scoping in functions

- A stack diagram keeps track of where each variable is defined, and its value.

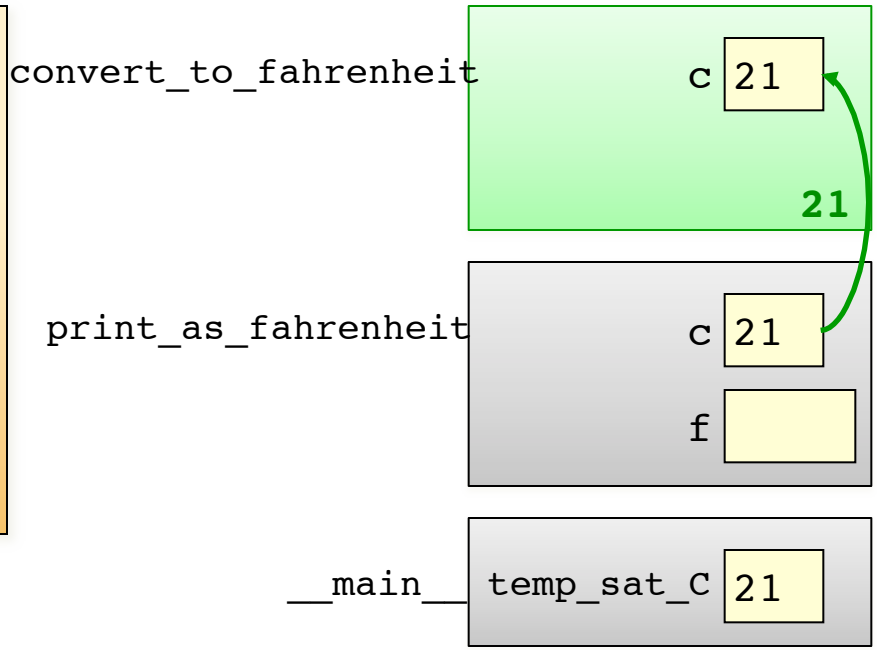
```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



Scoping in functions

- A stack diagram keeps track of where each variable is defined, and its value.

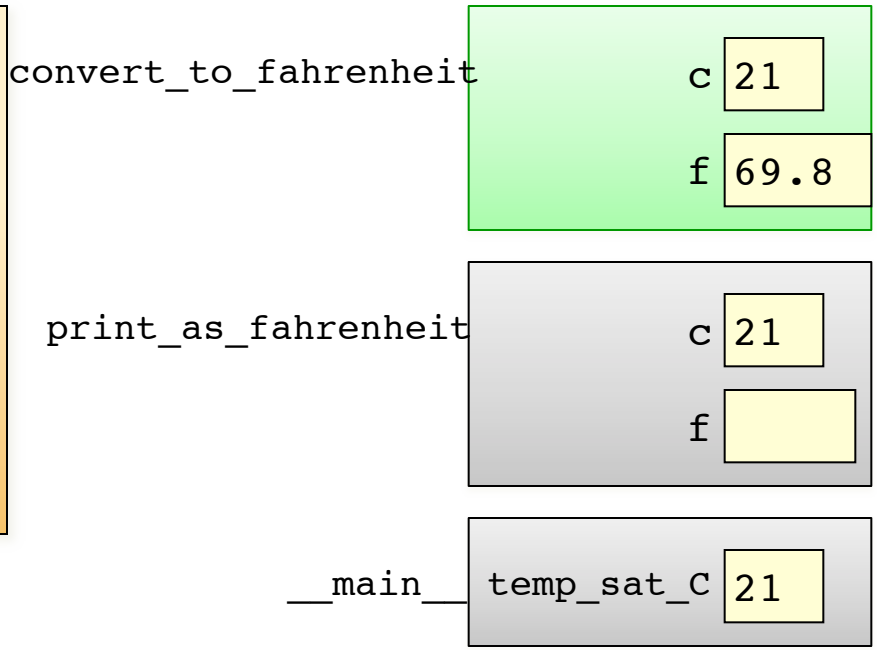
```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



Scoping in functions

- A stack diagram keeps track of where each variable is defined, and its value.

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

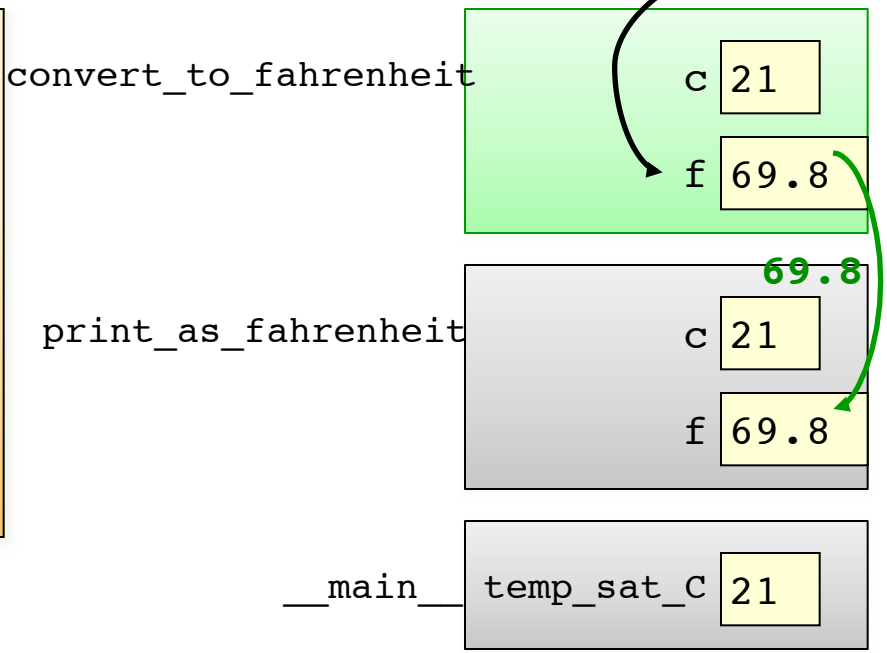


Scoping in functions

The return value is passed back to the function's caller.

- A stack diagram keeps track of where each variable is defined, and its value.

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

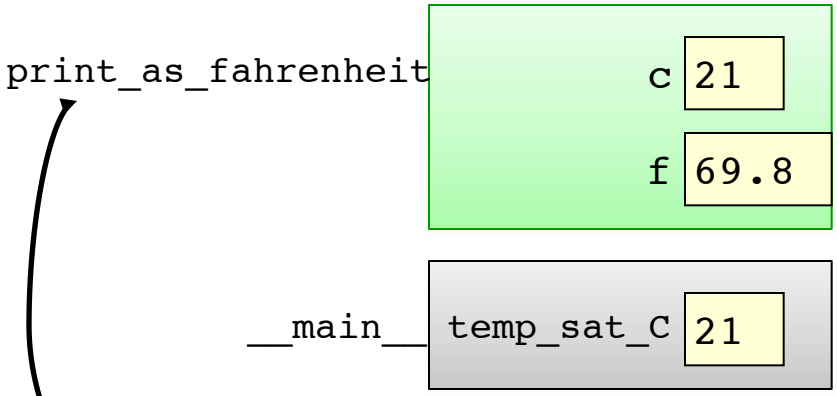


Scoping in functions

- A stack diagram keeps track of where each variable is defined, and its value.

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

When a function returns, its stack frame is popped off the stack.

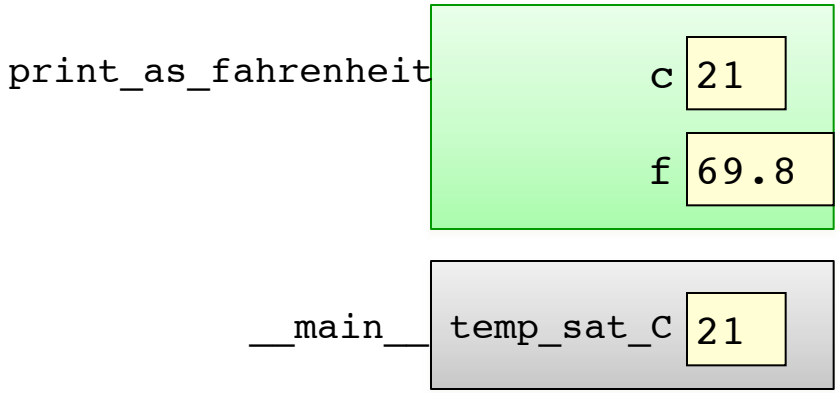


The stack frame for the calling function is now active again.

Scoping in functions

- A stack diagram keeps track of where each variable is defined, and its value.

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

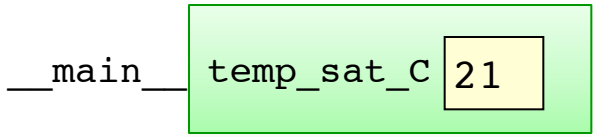



```
69.8 F
```

Scoping in functions

- A stack diagram keeps track of where each variable is defined, and its value.

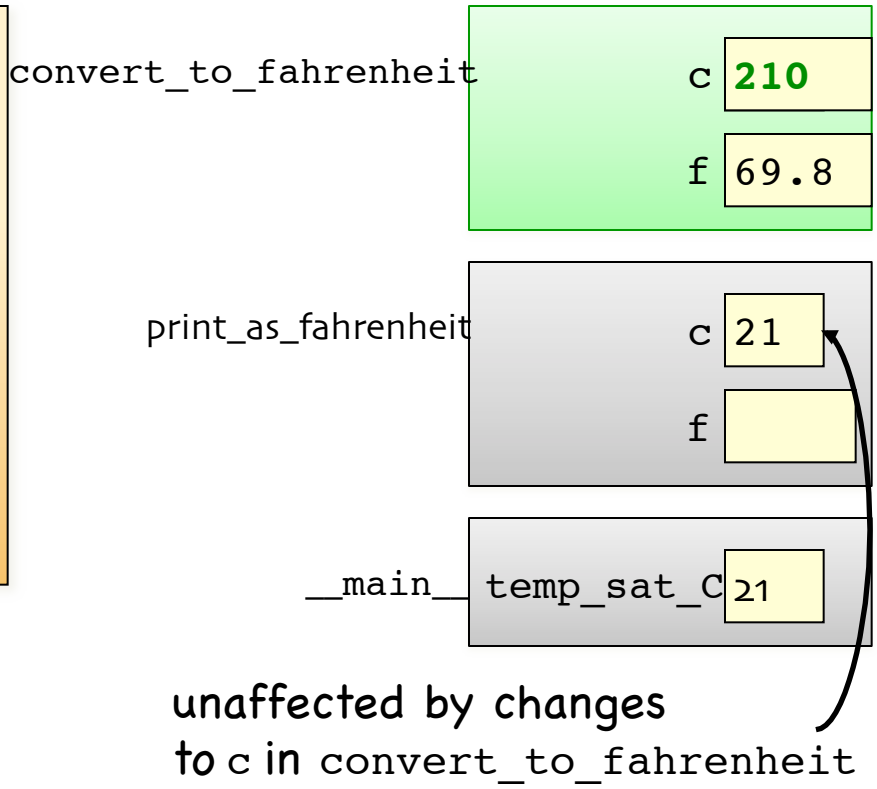
```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



Tricky issues with scoping

- Changes to a variable in the current scope do not affect variables in other scopes.

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    c = c * 10
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



Why use functions?

- Generalization: the same code can be used more than once, with parameters to allow for differences.

BEFORE

```
temp_sat_F = ((9.0 / 5.0) * 21) + 32.0
print 'Saturday:', temp_sat_F, 'F'

temp_sun_F = ((9.0 / 5.0) * 19) + 32.0
print 'Sunday:', temp_sun_F, 'F'

temp_mon_F = ((9.9 / 5.0) * 23) + 33.0
print 'Monday:', temp_mon_F, 'F'
```

Would not have made this typo.

AFTER

```
def print_as_fahrenheit(c, day):
    f = ((9.0 / 5.0) * c) + 32.0
    print day + ':', f, 'F'

print_as_fahrenheit(21, 'Saturday')
print_as_fahrenheit(19, 'Sunday')
print_as_fahrenheit(23, 'Monday')
```

Only type these lines once.

Why use functions?

- Maintenance: much easier to make changes.

BEFORE

```
temp_sat_F = ((9.0 / 5.0) * 21) + 32.0
print 'Saturday:', temp_sat_F, 'F'

temp_sun_F = ((9.0 / 5.0) * 19) + 32.0
print 'Sunday:', temp_sun_F, 'F'

temp_mon_F = ((9.9 / 5.0) * 23) + 33.0
print 'Monday:', temp_mon_F, 'F'
```

AFTER

```
def print_as_fahrenheit(c, day):
    f = ((9.0 / 5.0) * c) + 32.0
    print day + ':', f, 'F'

print_as_fahrenheit(21, 'Saturday')
print_as_fahrenheit(19, 'Sunday')
print_as_fahrenheit(23, 'Monday')
```

Can change to "Fahrenheit" with only one change.

Why use functions?

- Encapsulation: much easier to read and debug!

BEFORE

```
temp_sat_F = ((9.0 / 5.0) * 21) + 32.0
print 'Saturday:', temp_sat_F, 'F'

temp_sun_F = ((9.0 / 5.0) * 19) + 32.0
print 'Sunday:', temp_sun_F, 'F'

temp_mon_F = ((9.9 / 5.0) * 23) + 33.0
print 'Monday:', temp_mon_F, 'F'
```

What are we doing here?

Oh, printing as Fahrenheit!

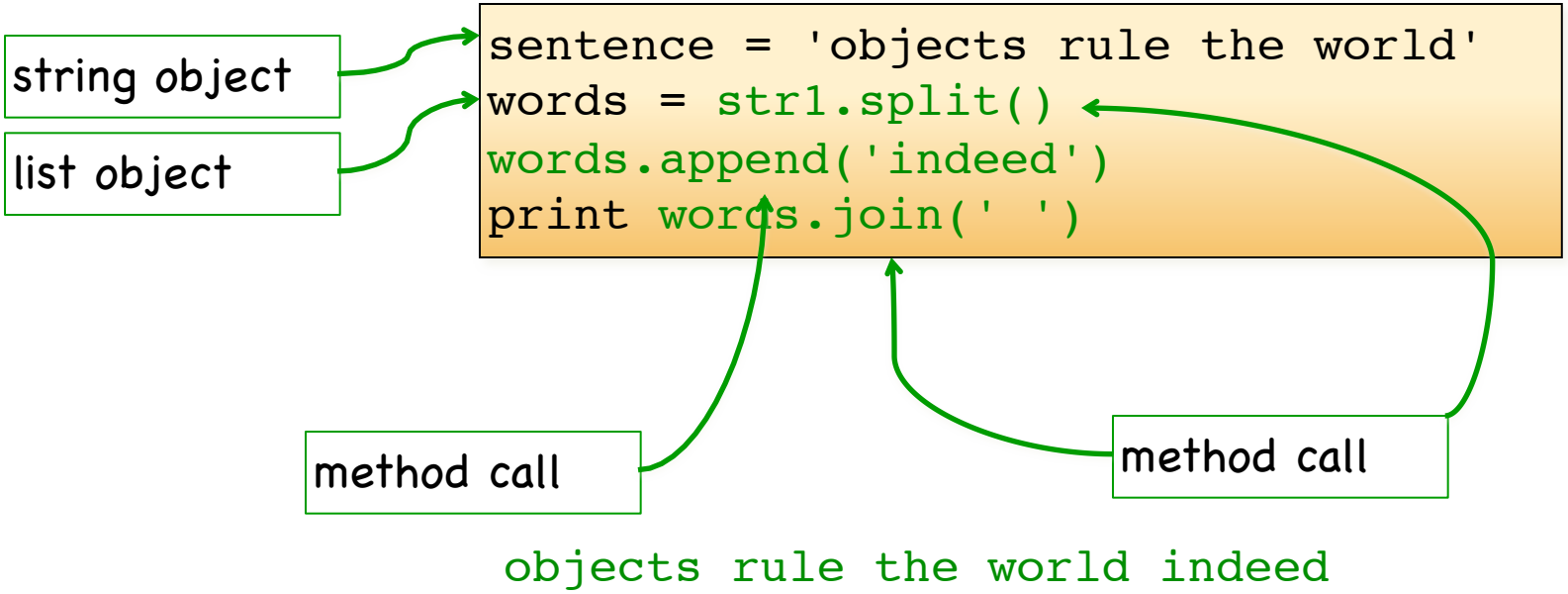
AFTER

```
def print_as_fahrenheit(c, day):
    f = ((9.0 / 5.0) * c) + 32.0
    print day + ':', f, 'F'

print_as_fahrenheit(21, 'Saturday')
print_as_fahrenheit(19, 'Sunday')
print_as_fahrenheit(23, 'Monday')
```

Using objects

- In Python everything is an object

Methods for string, list objects:



Defining a Class

```
class Car():
    wheels = 4

print Car.wheels
myCar = Car() #instantiation
print myCar.wheels #4
Car.wheels = 5 # change the class variable
print Car.wheels  #5
print myCar.wheels  #5
```

The Constructor

```
class Car():

    wheels = 4

    def __init__(self, color):
        self.color = color

#print Car.color <-- AttributeError: class Car has
   no attribute 'color'
myCar = Car("red")
print myCar.color # red
```

Adding Methods

```
class Car():
    wheels = 4
    def __init__(self, color):
        self.color = color
    def fade(self):
        self.color = self.color + "ish"

myCar = Car("red")
print myCar.color #red
myCar.fade()
print myCar.color #redish
```

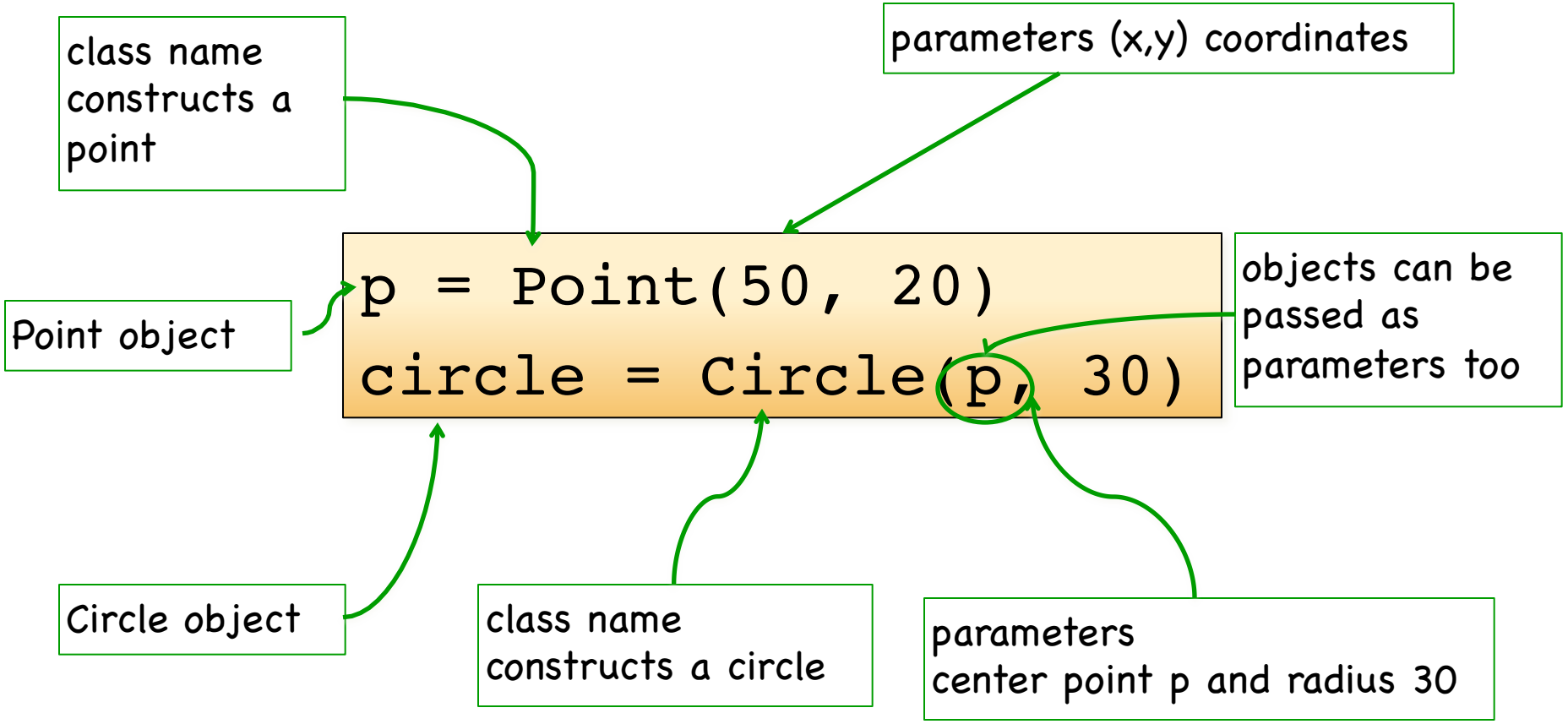
Inner Classes

```
class Car():
    wheels = 4
    def __init__(self, color, horsepower):
            self.color = color
            self.engine = self.Engine(horsepower)

    class Engine():
            def __init__(self, horsepower):
                self.horsepower = horsepower
            def getWatts(self):
                return self.horsepower * 745.7

myCar = Car('red', 400)
print myCar.engine.getWatts() #298280.0
```

Graphics Objects

- Use graphics.py module
- Graphics objects available:
  - Point
  - Line
  - Circle
  - Oval
  - Rectangle
  - Polygon
  - Text

Creating an object



Accessing Attributes and Methods

- Using dot (.)

```
p = Point(50, 20)
print p.x, p.y
print p.getX(), p.getY()
```

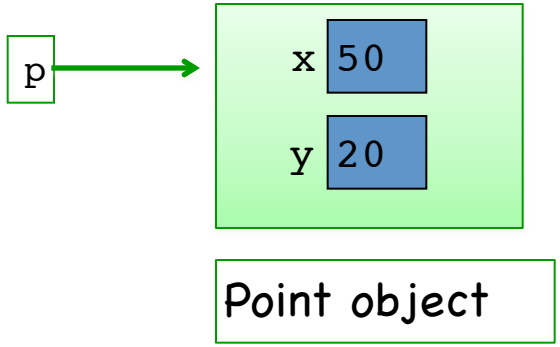
attributes or instance variables

methods to get the values of the entries

```
50 20
50 20
```

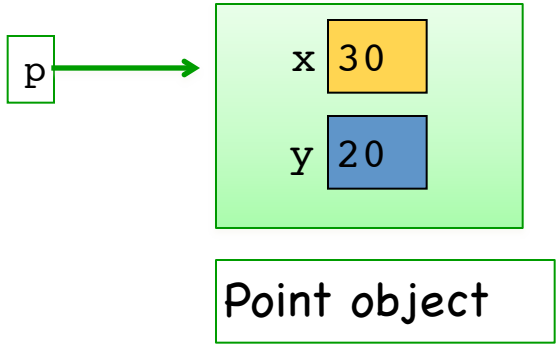
Objects are mutable

```
1  p = Point(50, 20)
2  p.x = p.x - 20
3  p2 = p
4  p2.x = p2.x + 10
5  print p.getX(), p.getY()
```



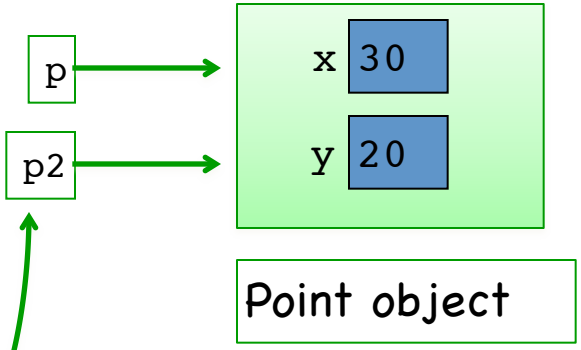
Objects are mutable

```
1  p = Point(50, 20)
2  p.x = p.x - 20
3  p2 = p
4  p2.x = p2.x + 10
5  print p.getX(), p.getY()
```



Objects are mutable

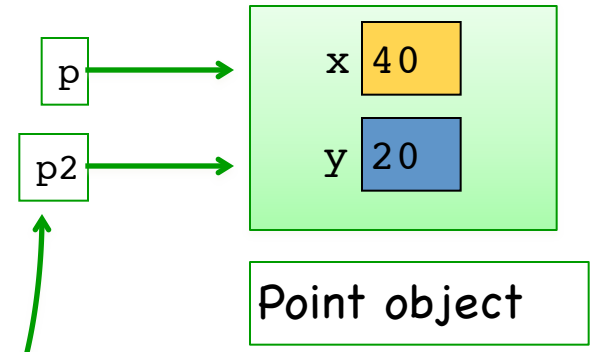
```
1  p = Point(50, 20)
2  p.x = p.x - 20
3  p2 = p
4  p2.x = p2.x + 10
5  print p.getX(), p.getY()
```



p2 is an alias of p, i.e. it refers to the same point object

Objects are mutable

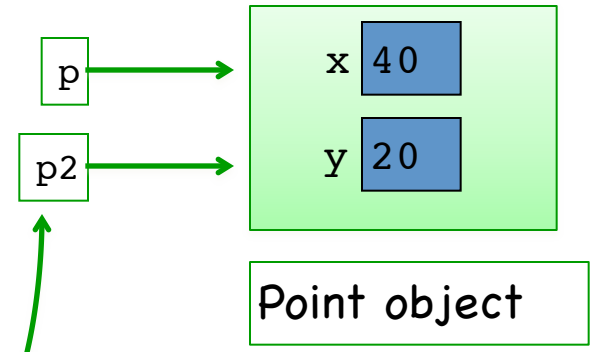
```
1  p = Point(50, 20)
2  p.x = p.x - 20
3  p2 = p
4  p2.x = p2.x + 10
5  print p.getX(), p.getY()
```



p2 is an alias of p, i.e. it refers to the same point object

Objects are mutable

```
1  p = Point(50, 20)
2  p.x = p.x - 20
3  p2 = p
4  p2.x = p2.x + 10
5  print p.getX(), p.getY()
```



p2 is an alias of p, i.e. it refers to the same point object

```
40 20
```

Simple Graphics Program

graphics module
defines the graphics objects
we will use

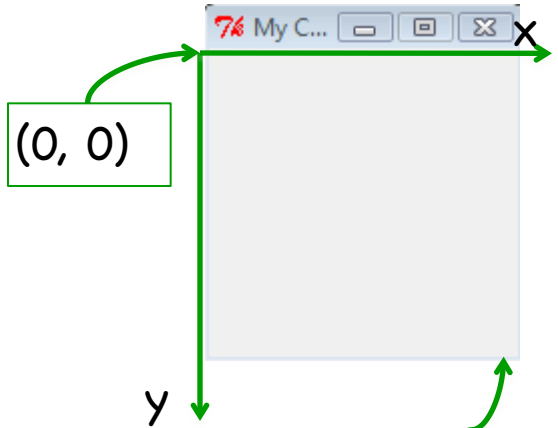
```
from graphics import *

win = GraphWin('My Circle', 100, 100)
c = Circle(Point(50,50), 10)
c.setFill('red')
c.draw(win)

win.mainloop()
```

Simple Graphics Program

```
from graphics import *

win = GraphWin('My Circle', 150, 150)
c = Circle(Point(50,50), 10)
c.setFill('red')
c.draw(win)

win.mainloop()
```

Creates a window with a canvas to draw on

Inverted coordinate system (units are pixels)



Window title

Canvas width

Canvas height

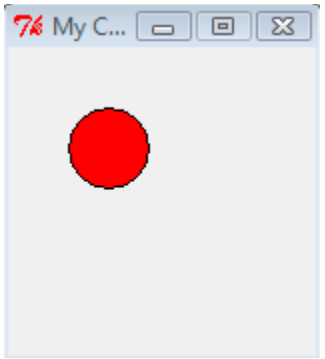
Simple Graphics Program

create a Circle object

```
from graphics import *

win = GraphWin('My Circle', 150, 150)
c = Circle(Point(50,50), 10)
c.setFill('red')
c.draw(win)

win.mainloop()
```

Circle center

Circle radius



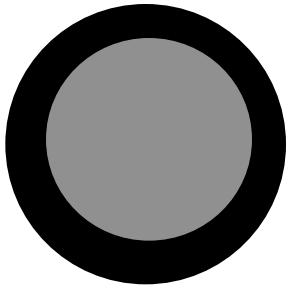
Simple Graphics Program

```
from graphics import *

win = GraphWin('My Circle', 150, 150)
c = Circle(Point(50,50), 10)
c.setFill('red')
c.draw(win)

win.mainloop()
```

every graphics program must end with this line;
it allows the window to process mouse clicks and keyboard input

User-defined types

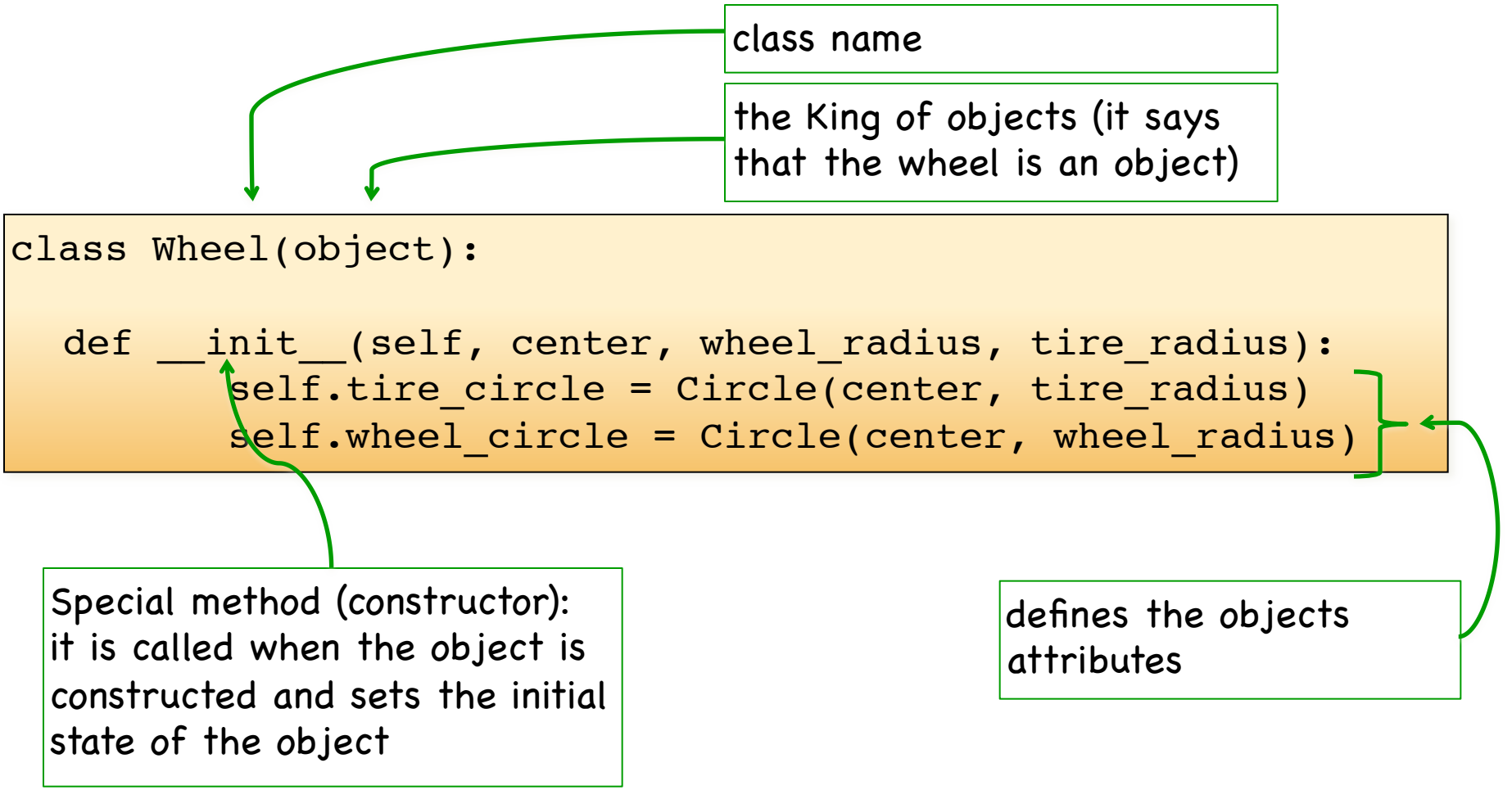
- What if we want to create our own class?
- E.g. let's create a class that draws a car wheel. For simplicity, the wheel will look like this:



Wheel class

- Attributes
  - `tire_circle`
  - `wheel_circle`
- Methods
  - `draw`
  - `move`
  - `get_size`
  - `get_center`
  - `set_color`

Wheel Class Definition



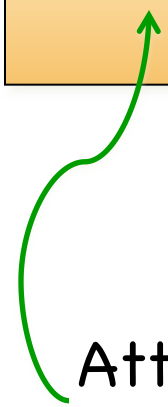
Wheel Class Definition

```
class Wheel(object):

  def __init__(self, center, wheel_radius, tire_radius):
        self.tire_circle = Circle(center, tire_radius)
        self.wheel_circle = Circle(center, wheel_radius)
```

- What is this `self` parameter?
- `self` is an alias to the object instance
- Must use it to access any of the object's attributes or methods
- it must always be the first parameter in a method signature

Wheel Class Definition

```
class Wheel(object):

  def __init__(self, center, wheel_radius, tire_radius):
      self.tire_circle = Circle(center, tire_radius)
      self.wheel_circle = Circle(center, wheel_radius)
```



Attributes are defined inside the __init__ method using the self parameter.

Attributes vs Local Variables

- Attribute
  - Defined in the __init__ method
  - Belongs to a specific object
  - Exists as long as the containing object exists
- Local variable
  - Declared within a method or a function
  - Exists only during the execution of its containing method or function

Wheel Class Definition

```
class Wheel(object):

  def __init__(self, center, wheel_radius, tire_radius):
        self.tire_circle = Circle(center, tire_radius)
        self.wheel_circle = Circle(center, wheel_radius)

  def draw(self, win):
      self.tire_circle.draw(win)
      self.wheel_circle.draw(win)

   def move(self, dx, dy):
      self.tire_circle.move(dx, dy)
      self.wheel_circle.move(dx, dy)
```

method definitions

Wheel Class Definition

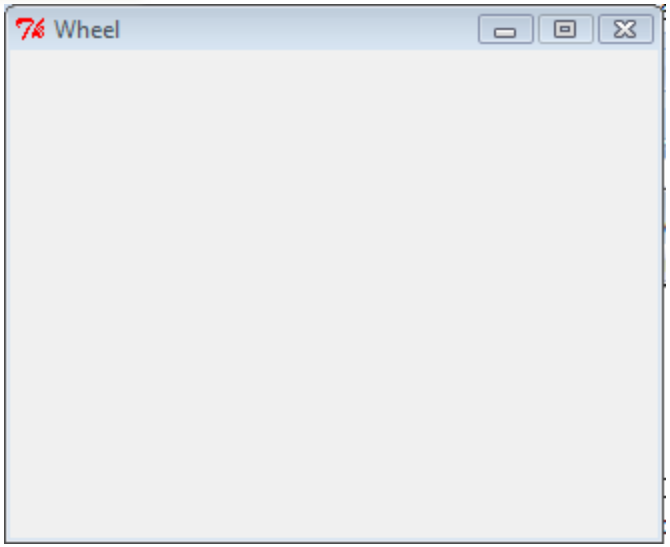
```
class Wheel(object):
   ''' This class defines a wheel template with two circles.
       Attributes: tire_circle, wheel_circle
    '''

   def __init__(self, center, wheel_radius, tire_radius):
         self.tire_circle = Circle(center, tire_radius)
         self.wheel_circle = Circle(center, wheel_radius)

   def draw(self, win):
         self.tire_circle.draw(win)
         self.wheel_circle.draw(win)

    def move(self, dx, dy):
         self.tire_circle.move(dx, dy)
         self.wheel_circle.move(dx, dy)

    def set_color(self, wheel_color, tire_color):
         self.tire_circle.setFill(tire_color)
         self.wheel_circle.setFill(wheel_color)
```

Wheel Class Definition

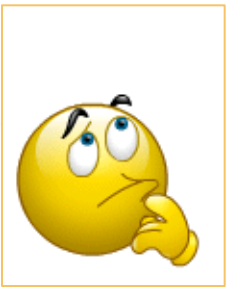
```
……...

def undraw(self):
      self.tire_circle.undraw()
      self.wheel_circle.undraw()

 def get_size(self):
      return self.tire_circle.getRadius()

def get_center(self):
  return tire_circle.getCenter()
```

Using our Wheel class

```
win = GraphWin('Wheel', 320, 240)
w = Wheel(Point(100, 100), 50, 70)
w.draw(win)
w.set_color('gray', 'black')
w.undraw()
win.mainloop()
```



Using our Wheel class

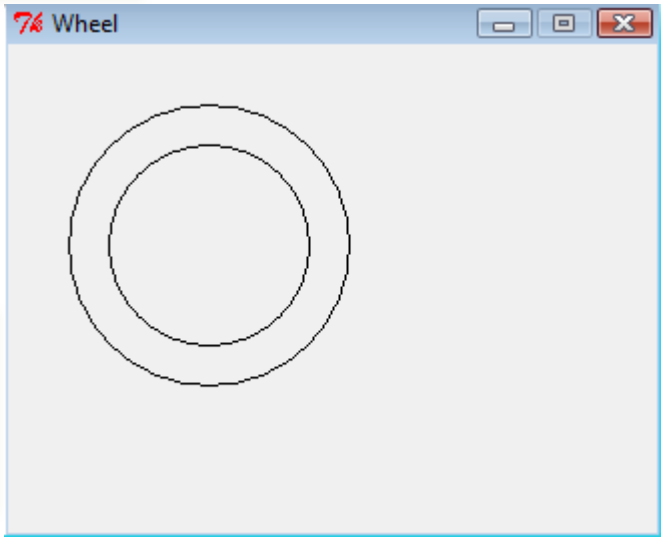


```
win = GraphWin('Wheel', 320, 240)
w = Wheel(Point(100, 100), 50, 70)
w.draw(win)
w.set_color('gray', 'black')
w.undraw()
win.mainloop()
```

What happened to the mysterious self parameter?

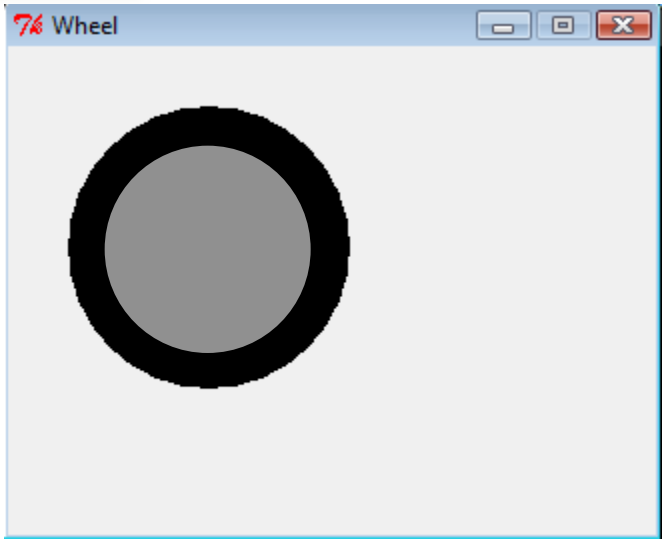
self = w

```
def draw(self, win):
   self.tire_circle.draw(win)
   self.wheel_circle.draw(win)
```



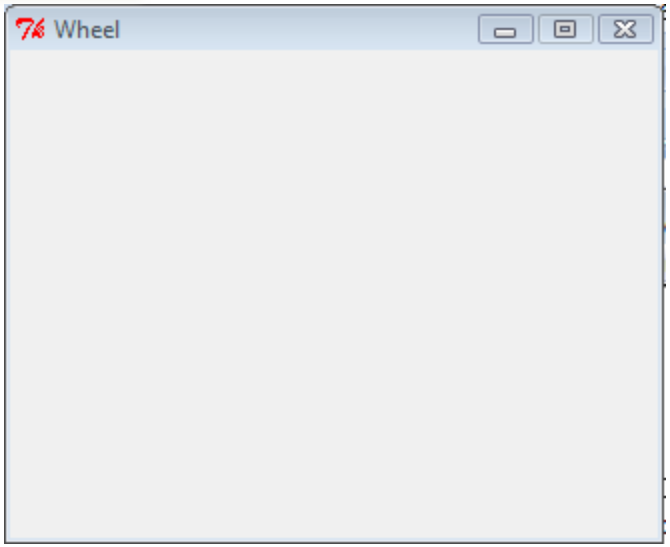
Using our Wheel class

```
win = GraphWin('Wheel', 320, 240)
w = Wheel(Point(100, 100), 50, 70)
w.draw(win)
w.set_color('gray', 'black')
w.undraw()
win.mainloop()
```



Using our Wheel class

```
win = GraphWin('Wheel', 320, 240)
w = Wheel(Point(100, 100), 50, 70)
w.draw(win)
w.set_color('gray', 'black')
w.undraw()
win.mainloop()
```



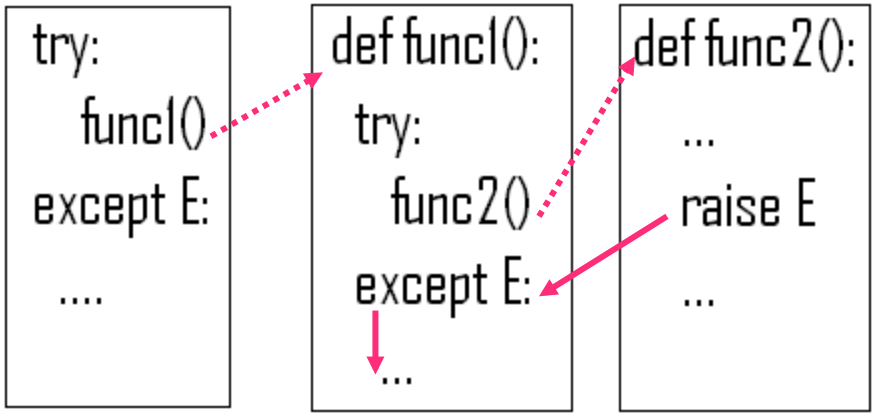
Exception Terminology

- **Exceptions** are events that can modify the flow or control through a program.
- **try/except** : catch and recover from the error raised by you or the Python interpreter
- **finally**: perform cleanup actions whether exceptions occur or not
- **raise**: trigger an exception manually in your code

Try, Except, Else and Finally

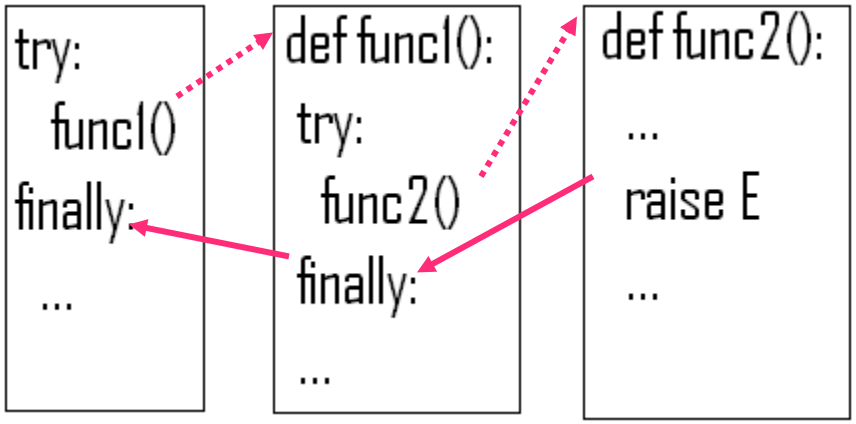
```
try:
    code to try

except pythonError1:
    exception code
except pythonError2:
    exception code
except:
     default except code

else:
    non exception case


finally:
    clean up code
```

Nesting Exception Handlers



Once the exception is caught, its life is over.

Nesting Exception Handlers



- But if the 'finally' block is present the code in the finally block will be executed, whether an exception gets thrown or not.

Raising Exceptions

```
try:
    raise NameError('HiThere')
except NameError:
    print 'An exception flew by!'
```

```
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

User Defined Exceptions

```
class MyError(Exception):

    def __init__(self, value):
        self.value = value

    def __str__(self):
        return repr(self.value)
```

User Defined Exceptions

```
try:
    raise MyError(2*2)
except MyError as e:
    print 'My exception occurred, value:', e.value
```

```
My exception occurred, value: 4
```

Substitution

```
>>> import re
>>> address = 'Ole Sangale Road'
>>> re.sub(’Road$', 'RD.',
  address)
’Ole Sangale RD.'
```

Reading a text file

- Easy in python:

```
For line in open(“asdf.txt”):
    print line
```

Efficient swapping of variables

- The normal way:

```
c=a
a=b
b=c
```

- The Python way:

```
a,b = b,a
```

  - More efficient – a temporary variable is never created.

Inline Conditionals

- You can do inline if/else statements to make simple coding shorter (similar to the “a ? b : c” concept in other languages)
- Ex:

```
Print “Equal” if A==B else “Not Equal”
```

Chained comparison operators

- Comparison operators can be chained:

```
X = 5
Return 1<x<10
```

Output: `True`

Step argument for slice operators

```
X = [1,2,3,4,5,6]
Print x[::2]   → [1,3,5]
Print x[::3]   → [1,4]
Print x[::-1]  → [6,5,4,3,2,1]
Print x[::-2]  → [6,4,2]
Print x[::-2][::-1]  → [2,4,6]
```

List comprehension

- Traditional for loop:

```
X = []
Y = [1,2,3,4,5,6]
for n in y:
    x.append(n**2)
```

- List Comprehension

```
X = [n**2 for n in y]
```

List Comprehensions

- They get even better:

```
[n**2 for n in x if n>3]
```

(only if n > 3)

```
[(n,n**2) for n in x]
```

(tuple with n and n^2)

List Comprehensions

- The Normal way:

```
mult_list = []
for a in [1,2,3,4]:
    for b in [5,6,7,8]:
        mult_list.append(a*b)
```

- The Python way:

```
mult_list= [a*b for a in [1,2,3,4] for b in [5,6,7,8]]
```

Program Organization

```
import MODULENAME

def func1():
   BODY1
...
def funcn(a):
   BODYN

class Class1(object):
   CLASSBODY1
...
class ClassN(object):
   CLASSBODYN

# start of the program
MAINBODY
```