Accelerating Information Technology Innovation

http://aiti.mit.edu/program/philippines-summer-2012/

Philippines Summer 2012
Lecture 2 – Intermediate Python
June 21, 2012

MISTI

Google

MIT

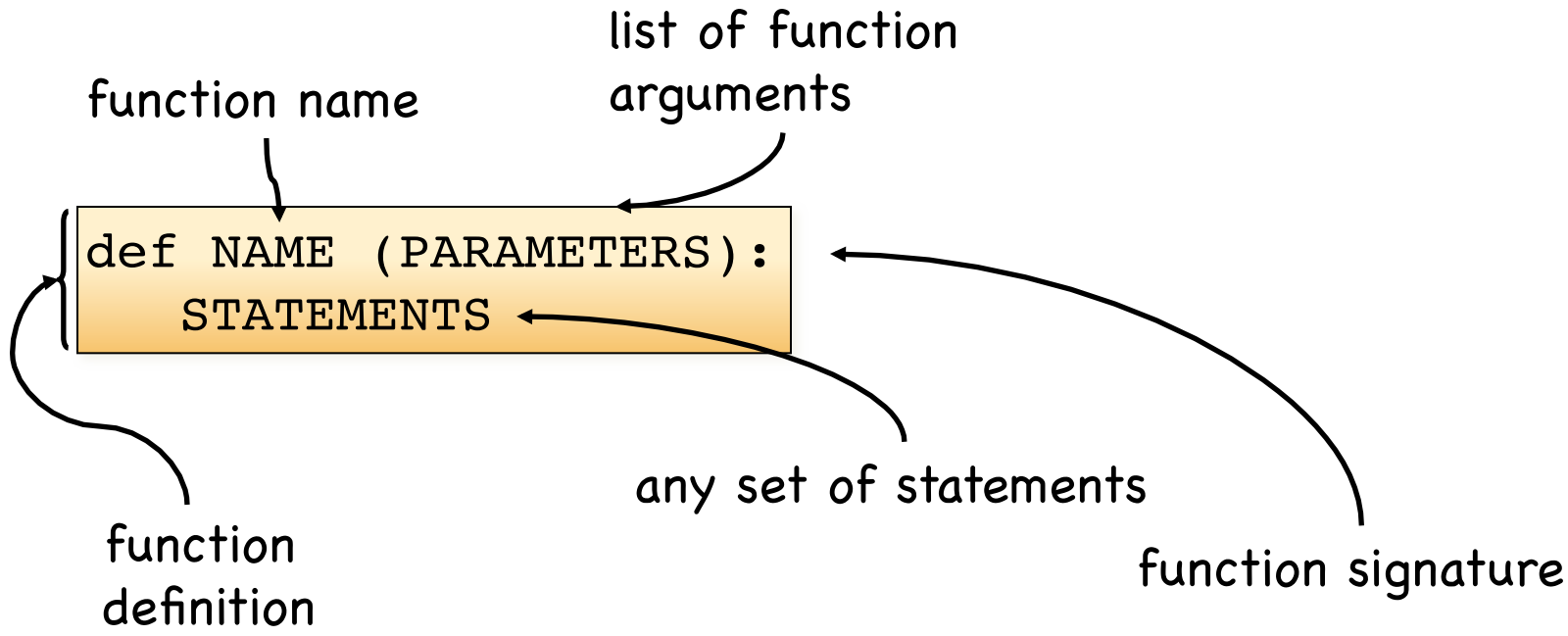
Agenda

- More on Functions
- Object Oriented Programming
- Exceptions
- Regular Expressions
- How to be a Python Ninja!

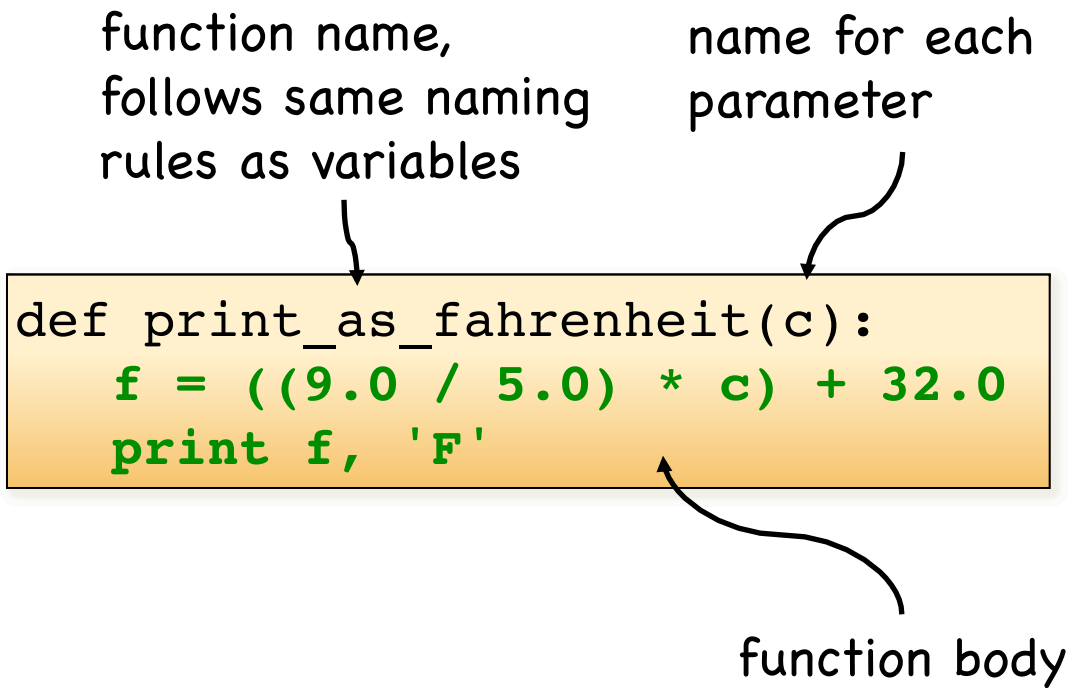
Functions

Functions

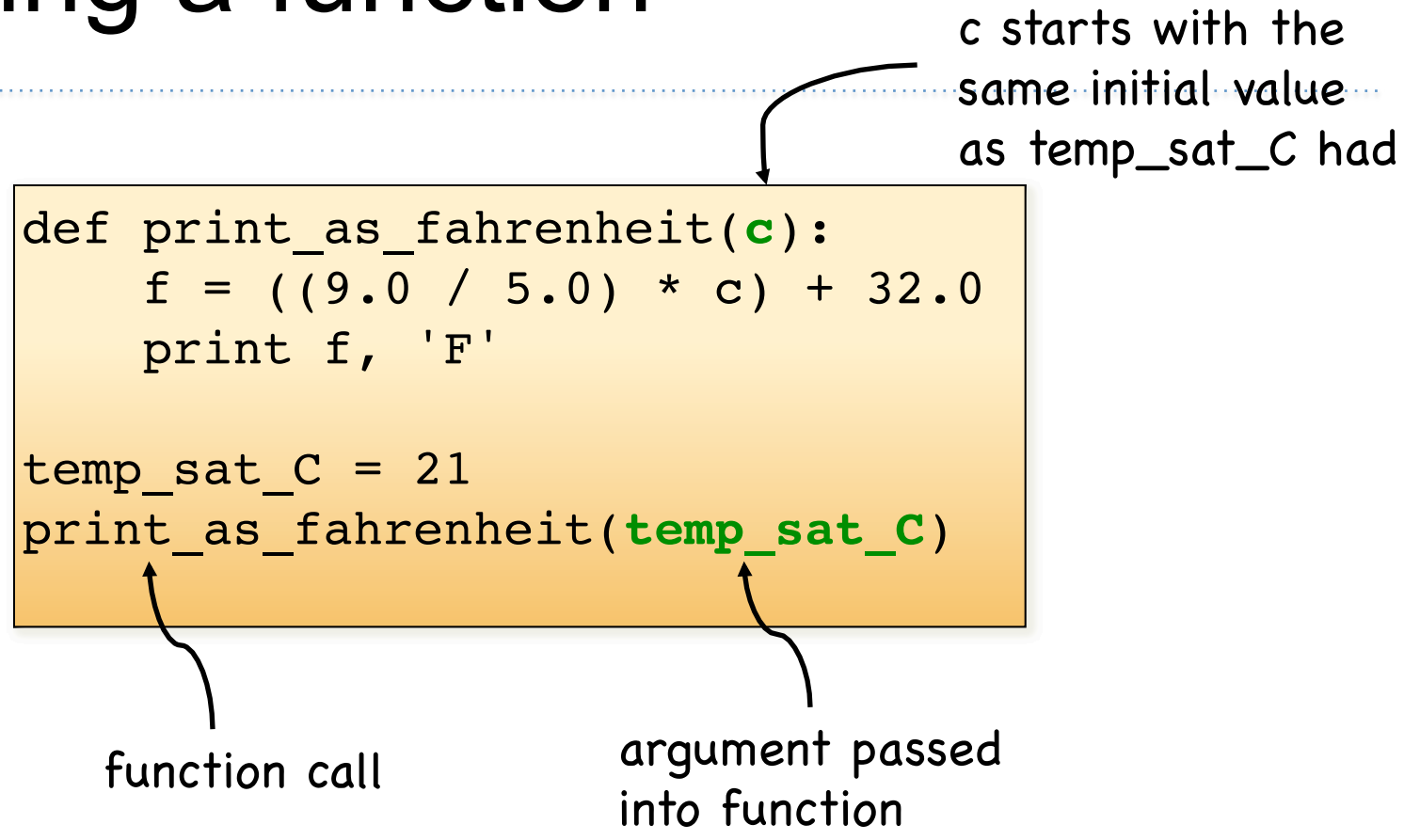
- A function is a sequence of statements that has been given a name.



Defining a function

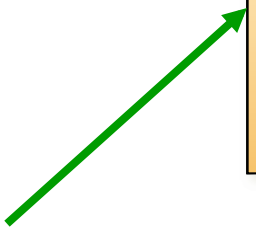


Calling a function



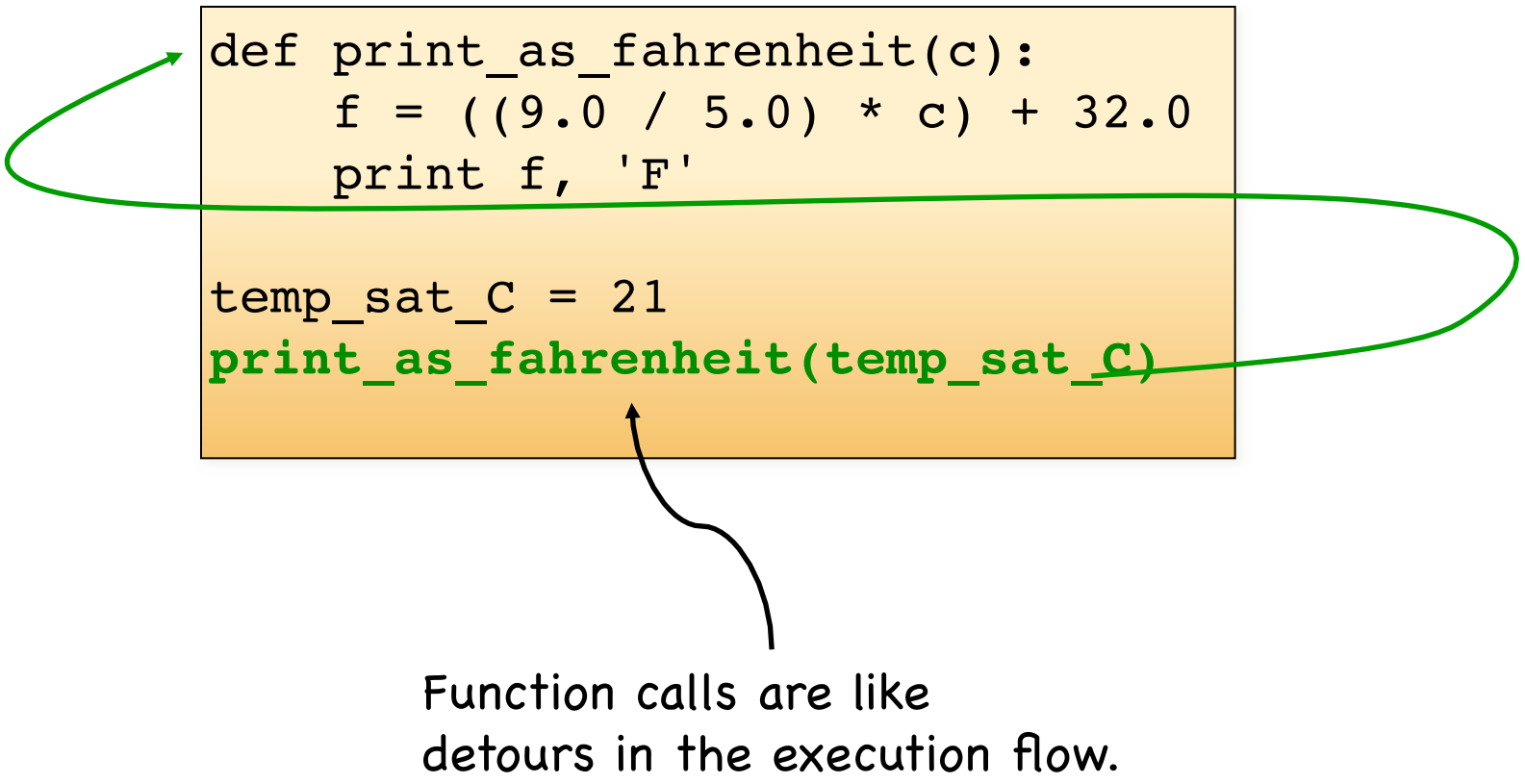
Flow of execution

```
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

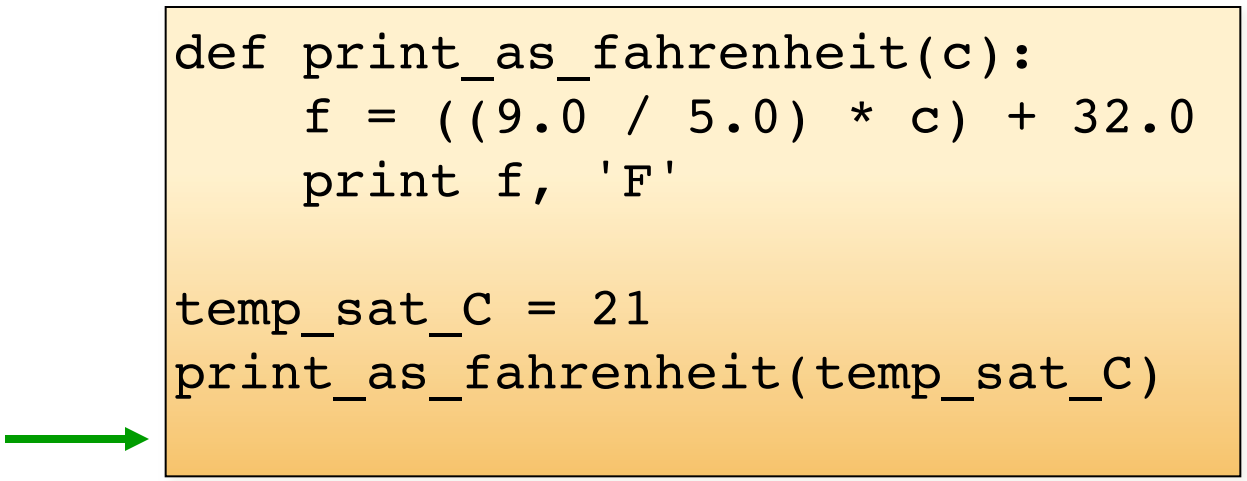
```
def print_as_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    print f, 'F'

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
```

Different numbers of parameters

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

```
print_as_fahrenheit(21, 'Saturday')
```

```
Saturday: 69.80000000000001 F
```

```
def print_forecast_intro():
    print 'Welcome to your weather forecast!'
```

```
print_forecast_intro()
```

```
Welcome to your weather forecast!
```

Different numbers of parameters

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

- What happens here?

```
print_as_fahrenheit(21)
```

```
TypeError: print_as_fahrenheit() takes exactly 2 arguments (1 given)
```

```
print_as_fahrenheit(21, 'Saturday', ‘Sunday’)
```

```
TypeError: print_as_fahrenheit() takes exactly 2 arguments (3 given)
```

```
print_as_fahrenheit('Saturday', 21)
```

```
TypeError: can't multiply sequence by non-int of type 'float'
```

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

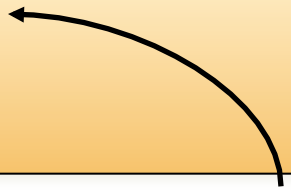
What is the output here?

```
def convert_to_fahrenheit(c):
    print ‘Celsius:’ + c
    f = ((9.0 / 5.0) * c) + 32.0
    return f
    print ‘Fahrenheit:’ + f

convert_to_fahrenheit(27)
```

```
Celsius: 27
```

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
```

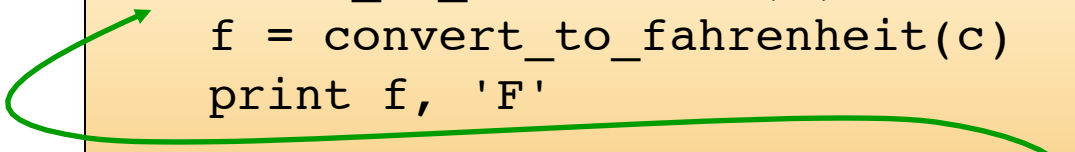
Good rule: Every path through the function must have a return statement.
If you don't add one, Python will add one for you that returns nothing (the value None).

Functions can call functions

```
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)
```

Functions can call functions

```
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)
```



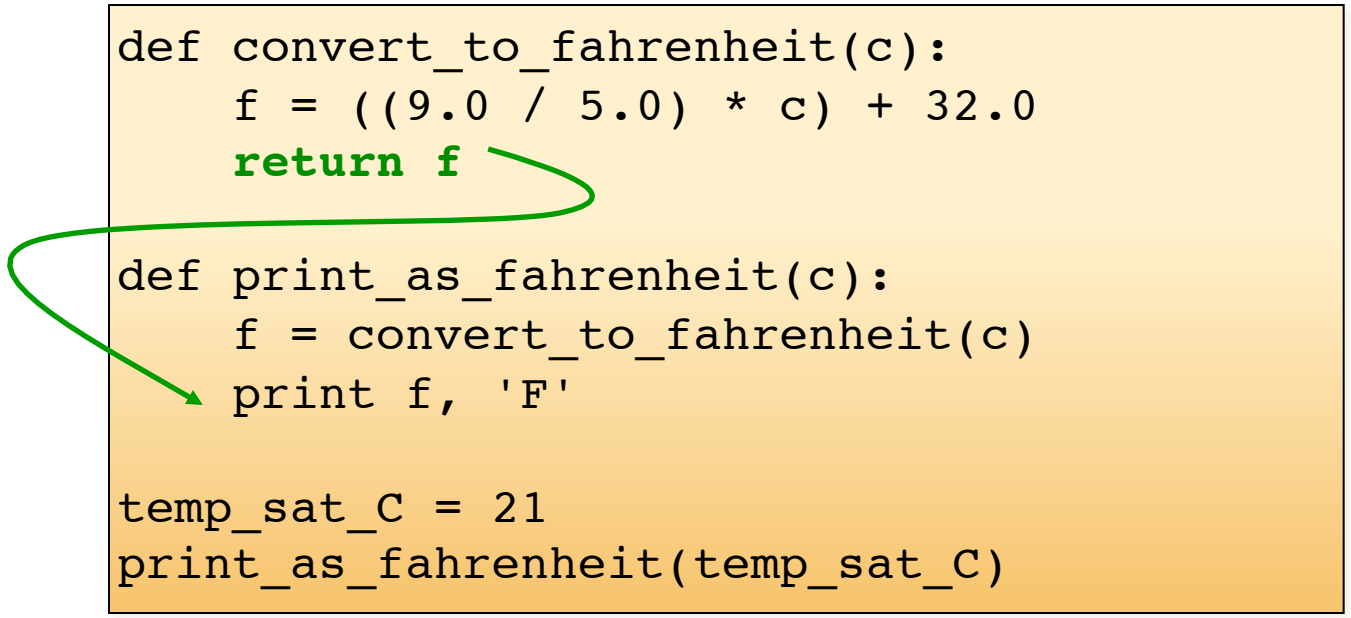
Functions can call functions

```
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)
```

Functions can call functions

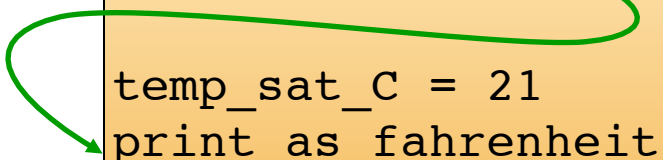
```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

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

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

Functions can call functions



Functions can call functions

```
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)
```



Functions can call functions

```
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)
```

```
80.6 F
```

What is wrong here?

this function has to be defined before it is called

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

```
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
```

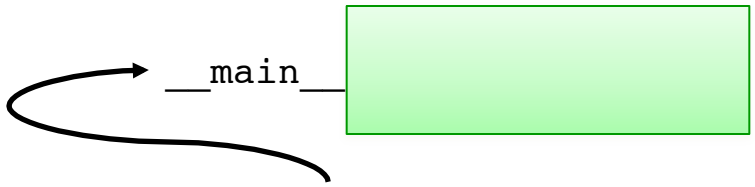
what about this one?

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)
```



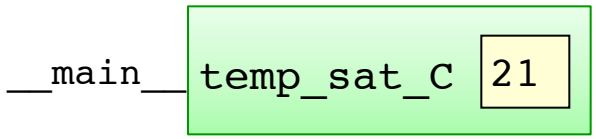
Each function call gets its own stack frame.

26

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)
```

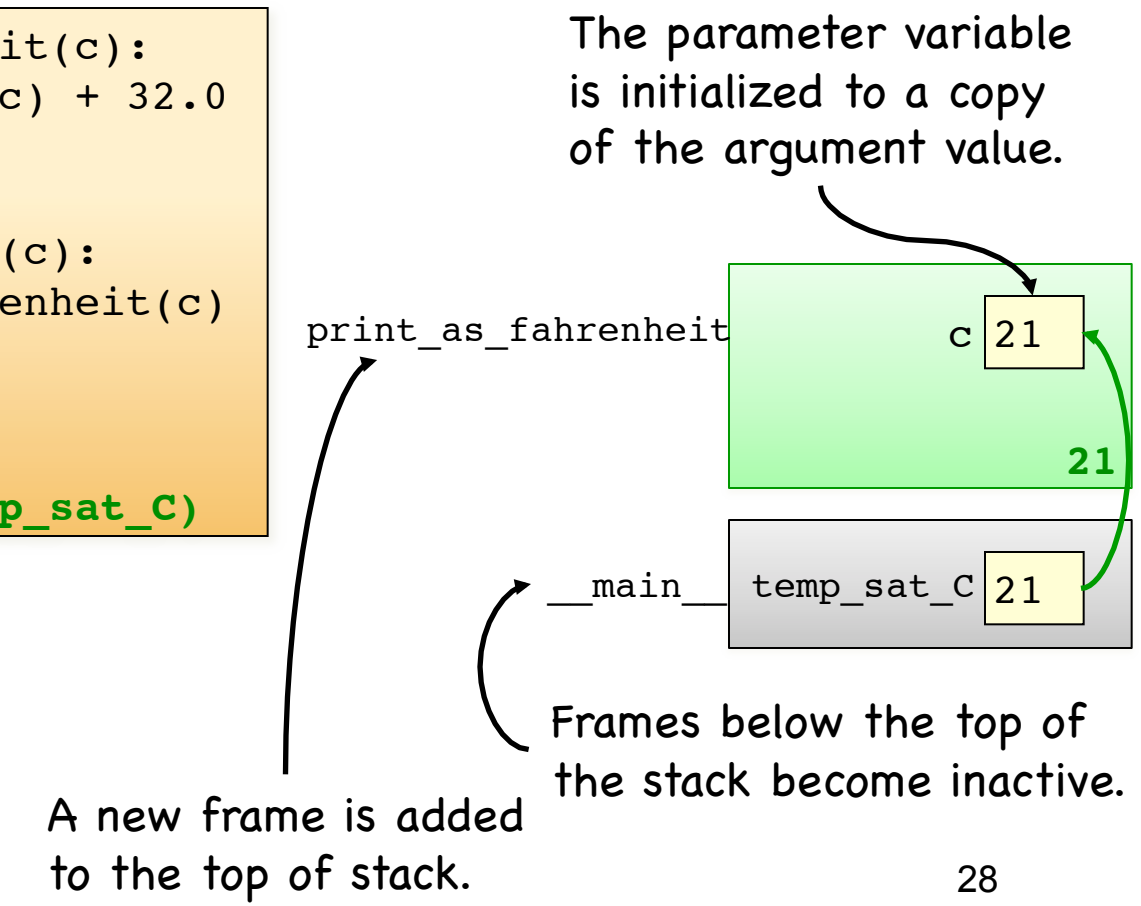


27

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)
```



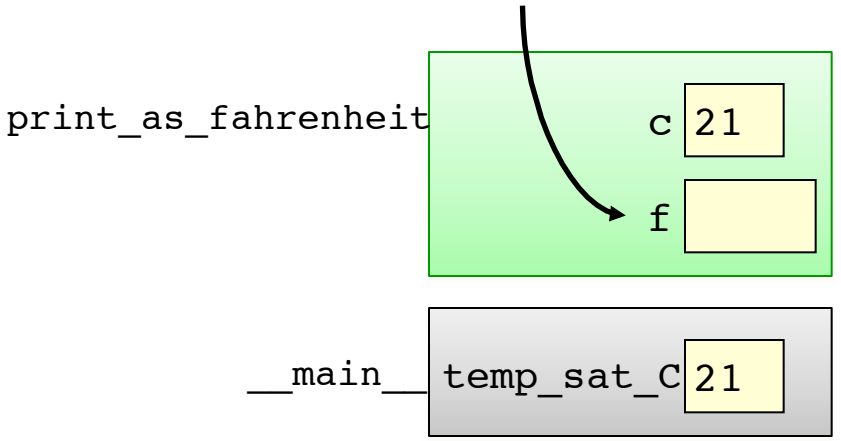
28

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)
```

Variables defined inside the function are called local variables.

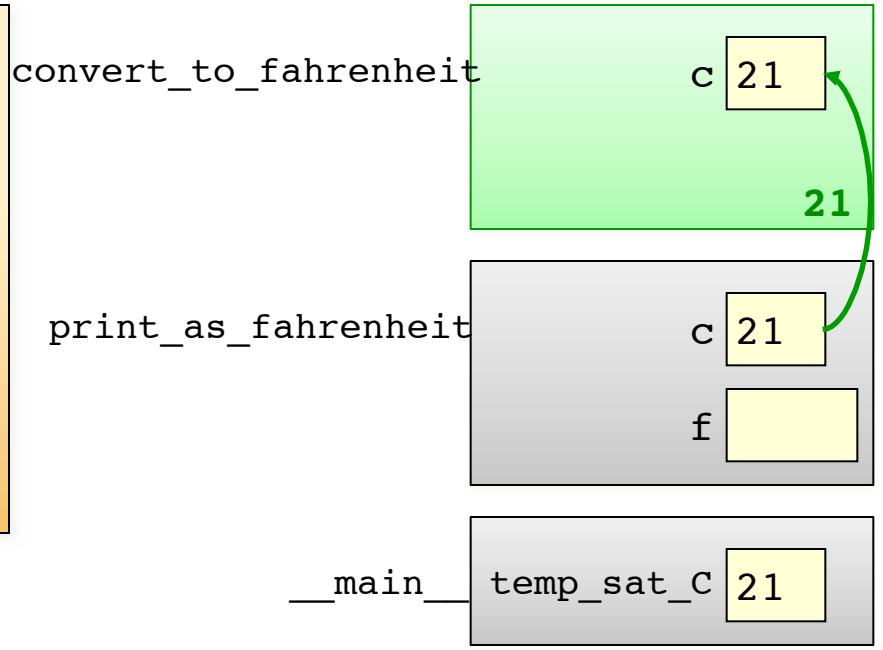


29

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)
```

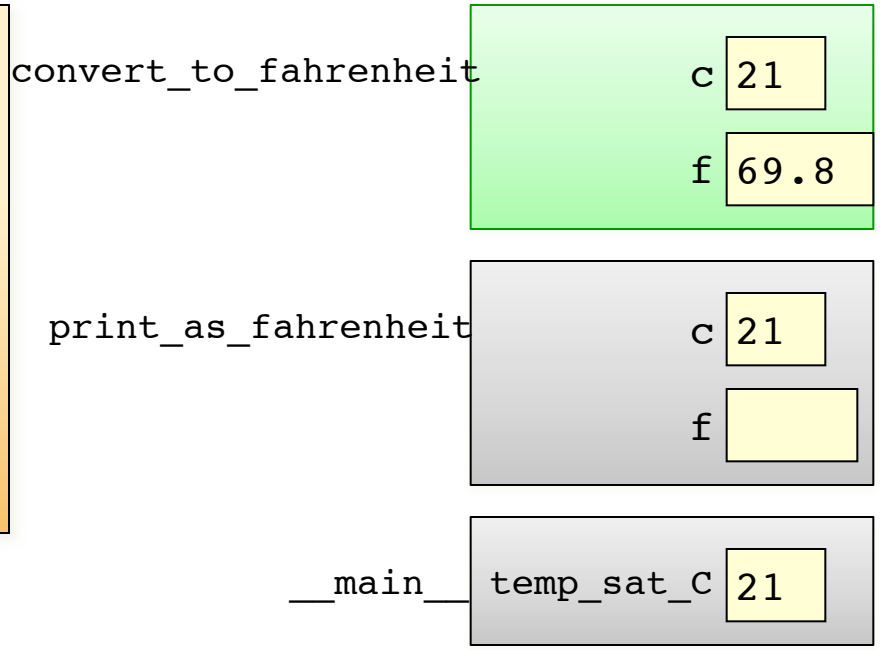


30

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)
```

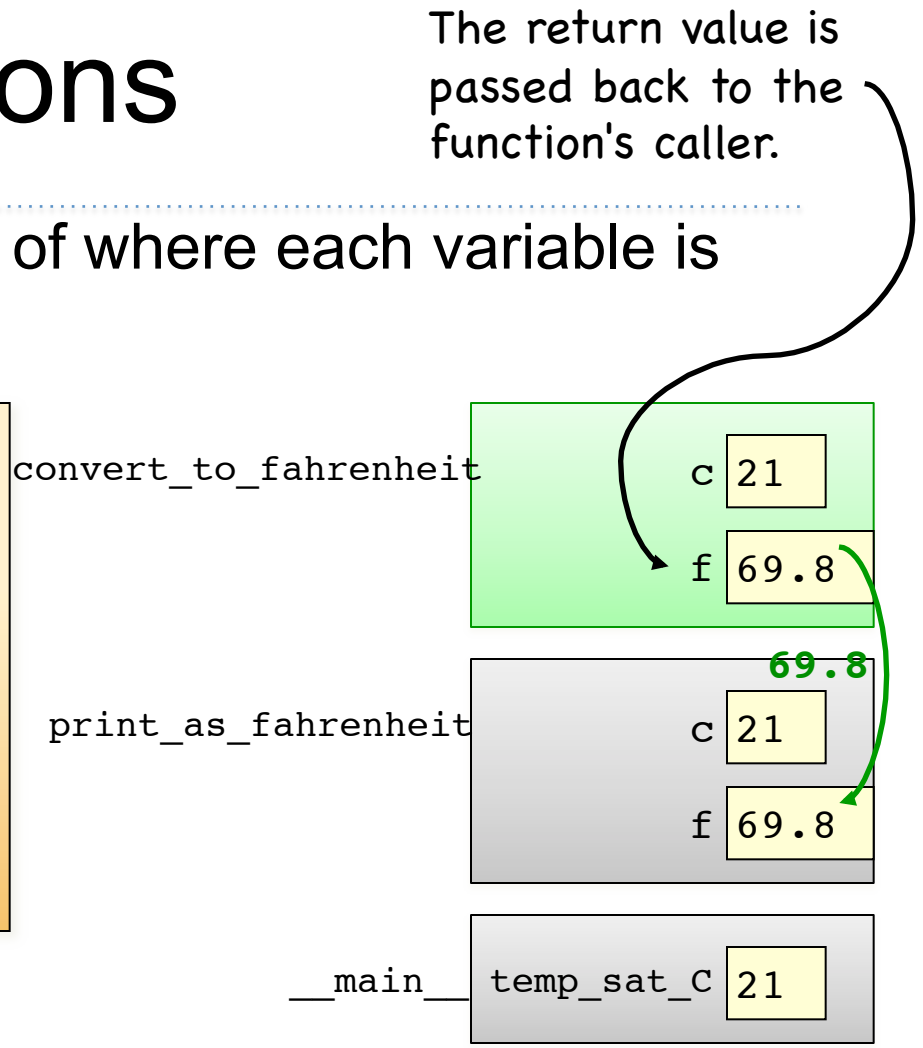


31

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)
```



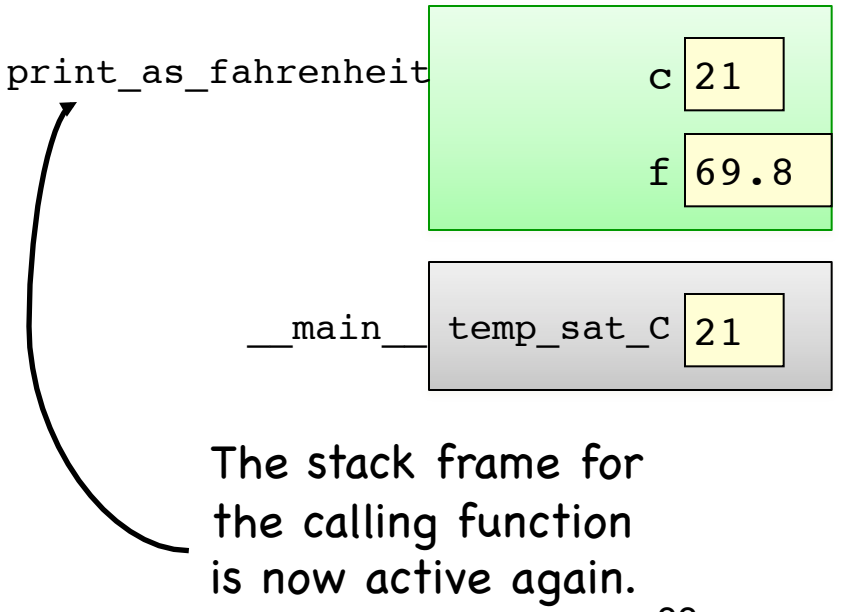
32

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.



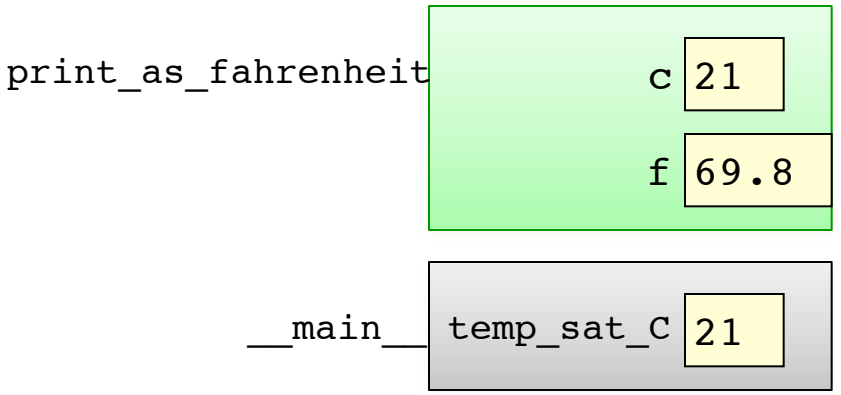
The stack frame for the calling function is now active again.

33

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)
```



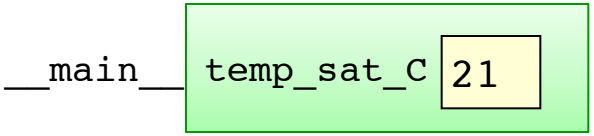

```
69.8 F
```


34

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)
```

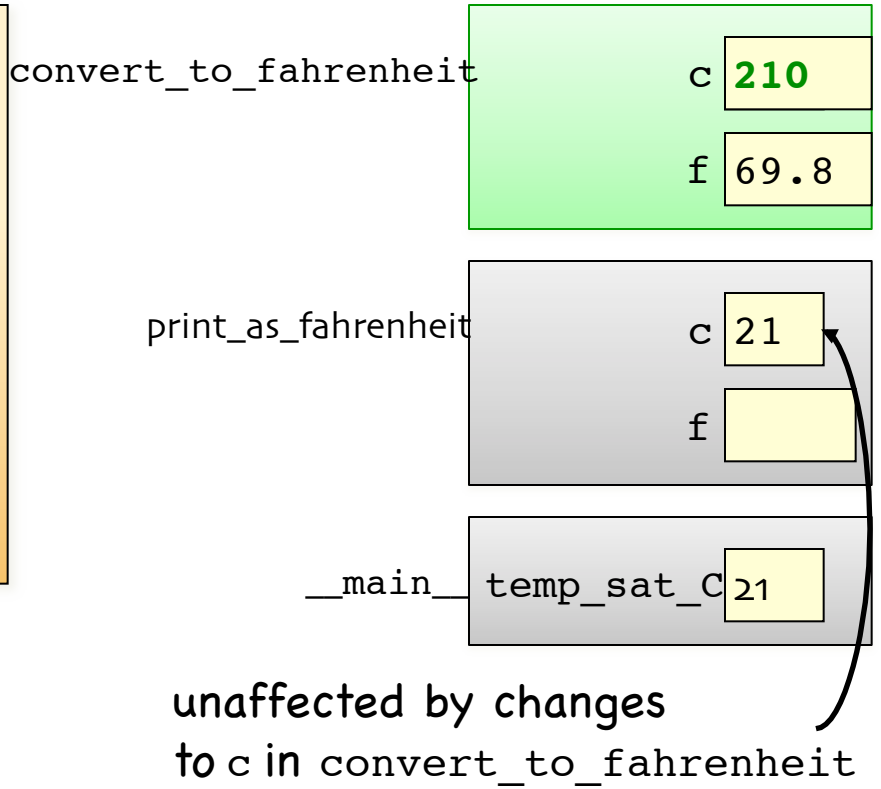


35

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)
```



36

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.

37

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.

38

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')
```


39

Object Oriented Programming

The History of Objects

- Objects weren't always supported by programming languages
- Idea first originated at MIT in the 1960s and was officially incorporated in a few languages in the same decade
- OOP (Object Oriented Programming) has now become a core feature of nearly all languages

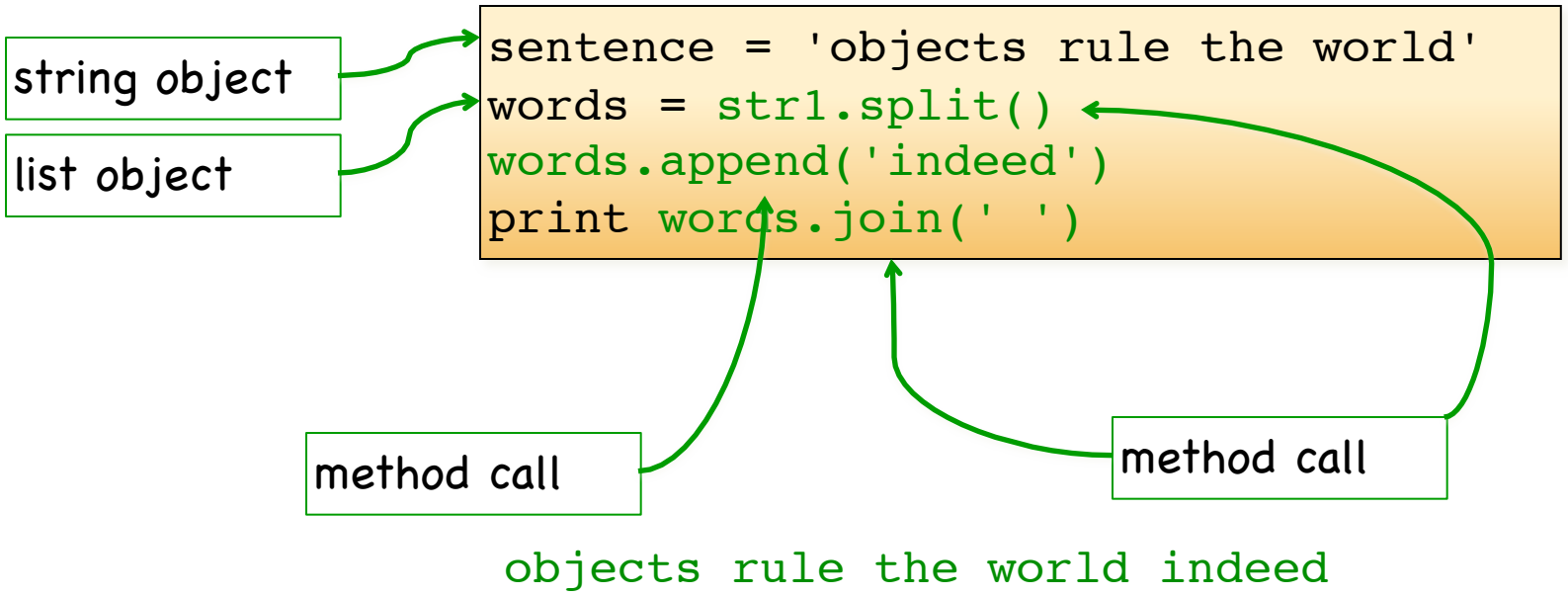
Object Oriented Programming (OOP)

- A certain style of computer programming centered around data structures called “objects”
- Objects are a standard way to organize data

Using objects

- In Python everything is an object

Methods for string, list objects:



43

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
```

Static Methods

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

    @staticmethod
    def isOld(miles):
        if miles < 50000:
            return False
        return True

print Car.isOld(30000) # False
```

Inner Classes

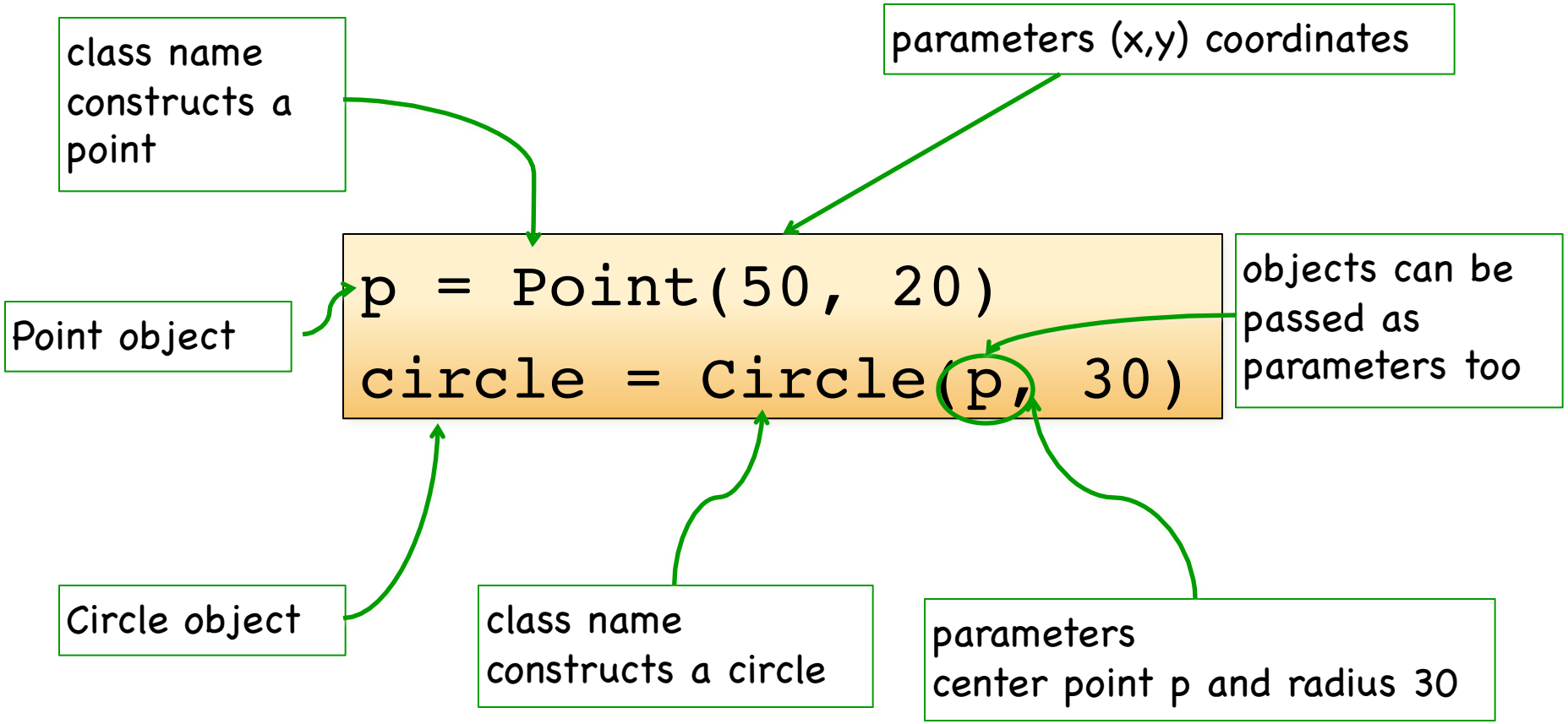
```
class Car():
    wheels = 4
    def __init__(self, color, horsepower):
            self.color = color
            self.engine = self.Engine(horsepower)

    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

49

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

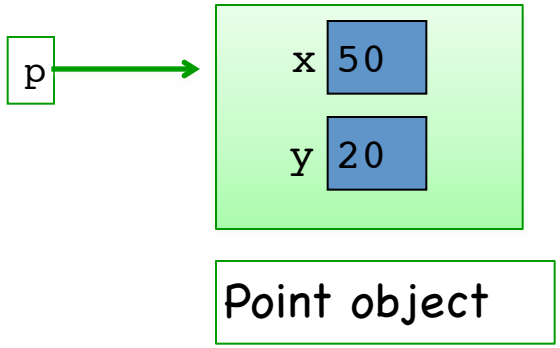
methods to get the values of the entries

```
50 20
50 20
```


51

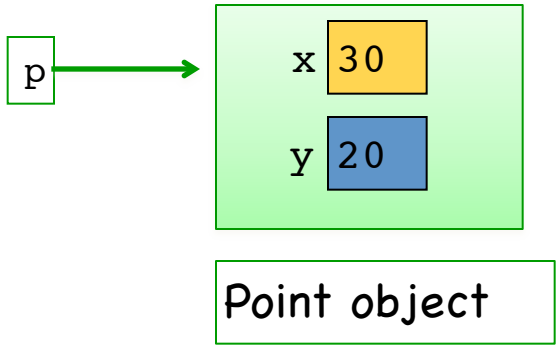
Objects are mutable

```
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()
```



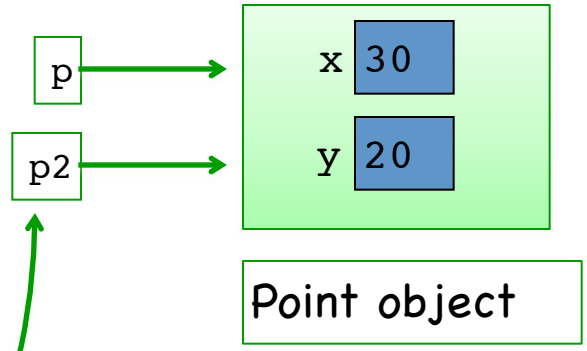
Objects are mutable

```
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

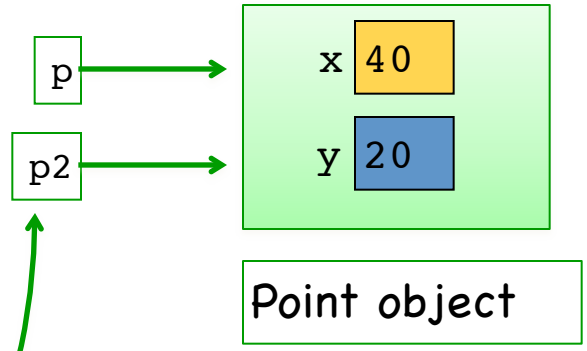
```
1  p = Point(50, 20)
2  p.x = p.x - 20
3  p2 = p
4  p2.x = p2.x + 10
5  print p.getX(), p.getY()
```



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

Objects are mutable

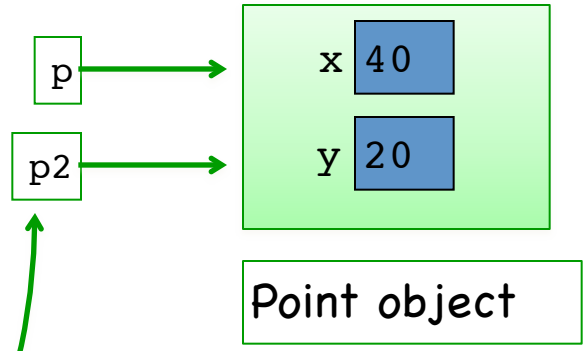
```
1  p = Point(50, 20)
2  p.x = p.x - 20
3  p2 = p
4  p2.x = p2.x + 10
5  print p.getX(), p.getY()
```



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
```

Scoping in functions

- Basic types - create a copy of the variable inside the function

```
def move_by_10(x, y):
  x = x + 10
  y = y + 10

x = 10
y = 10
move_by_10(x, y)
print x, y
```

What does this print?

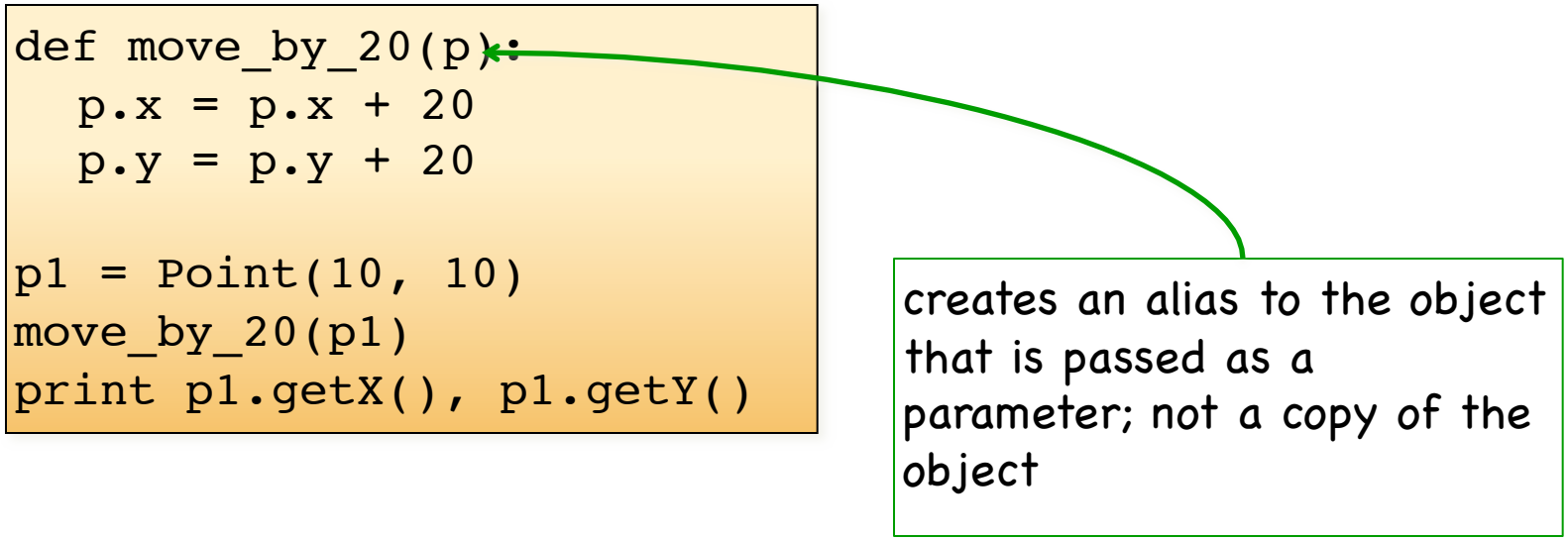
```
10 10
```


57

Scoping in functions

- Objects - create an alias of the variable inside the function

```
def move_by_20(p):
  p.x = p.x + 20
  p.y = p.y + 20

p1 = Point(10, 10)
move_by_20(p1)
print p1.getX(), p1.getY()
```



What does this print?

30 30

58

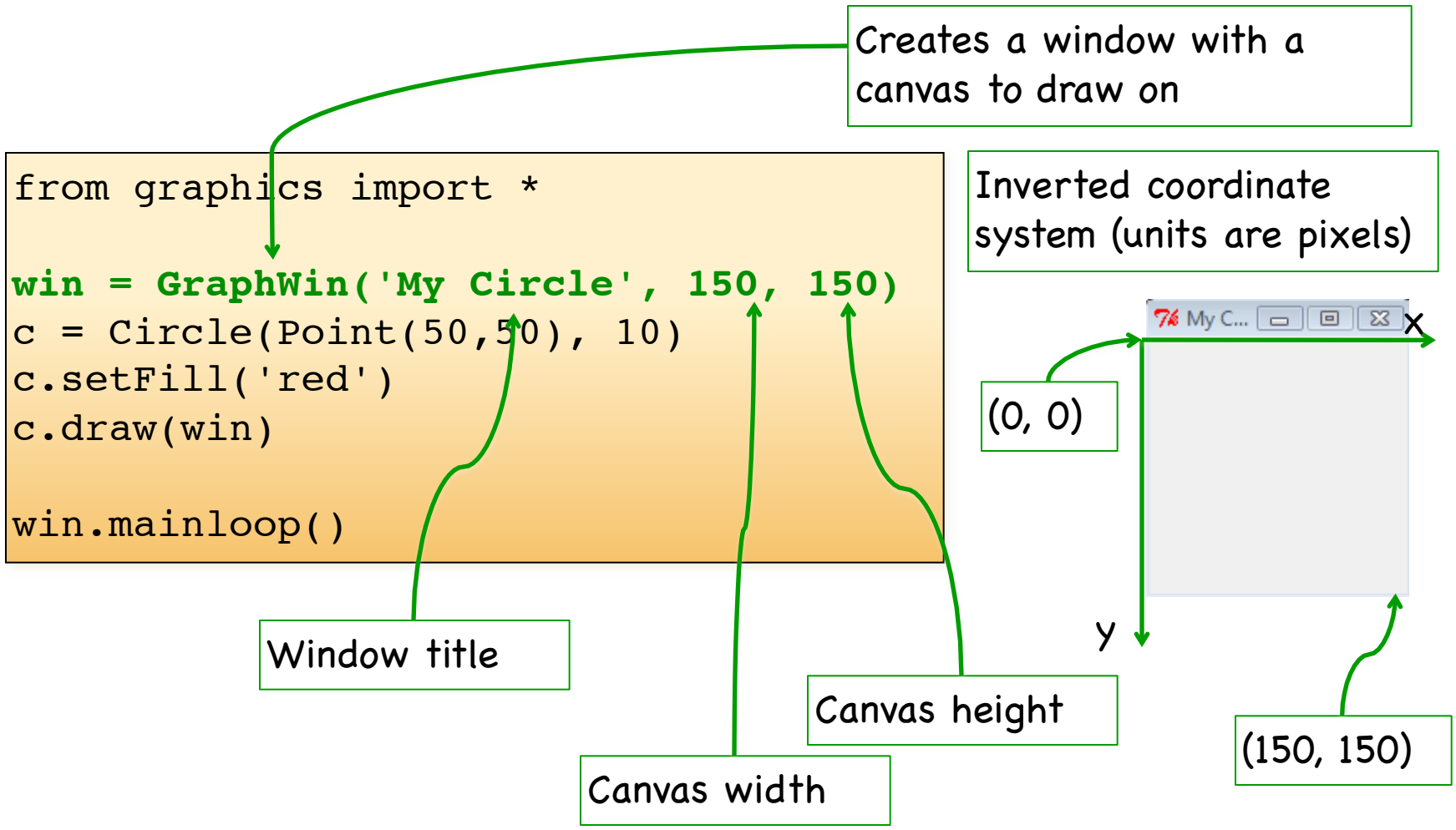
Simple Graphics Program

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()
```


59

Simple Graphics Program



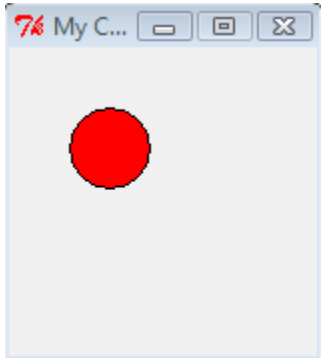
Simple Graphics Program

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

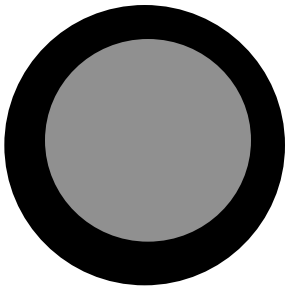
```
from graphics import *

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

62

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:



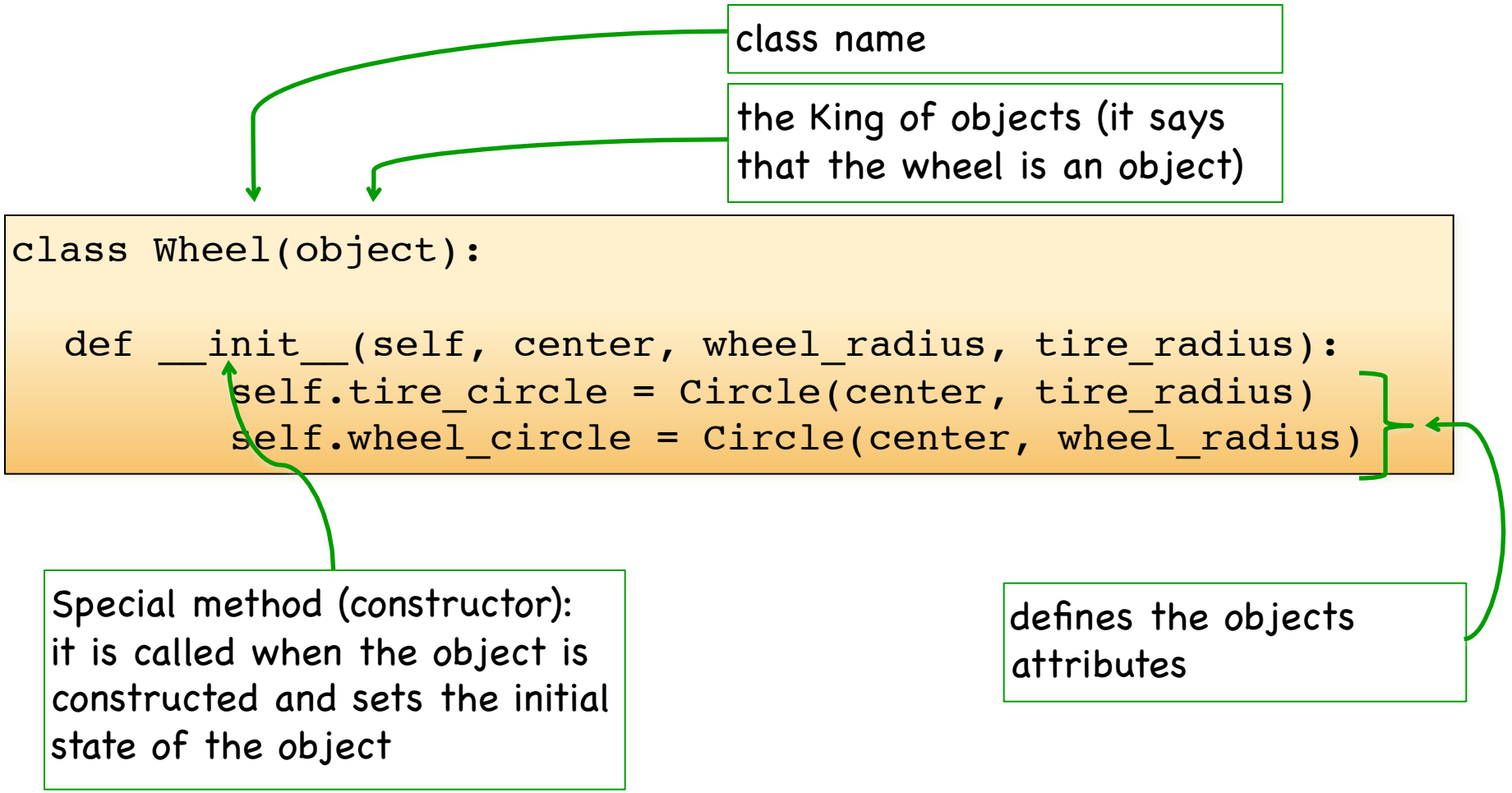
63

Wheel class

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

64

Wheel Class Definition



Wheel Class Definition

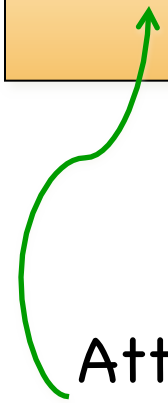
```
class Wheel(object):

  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

66

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.

67

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

68

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

69

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)
```


70

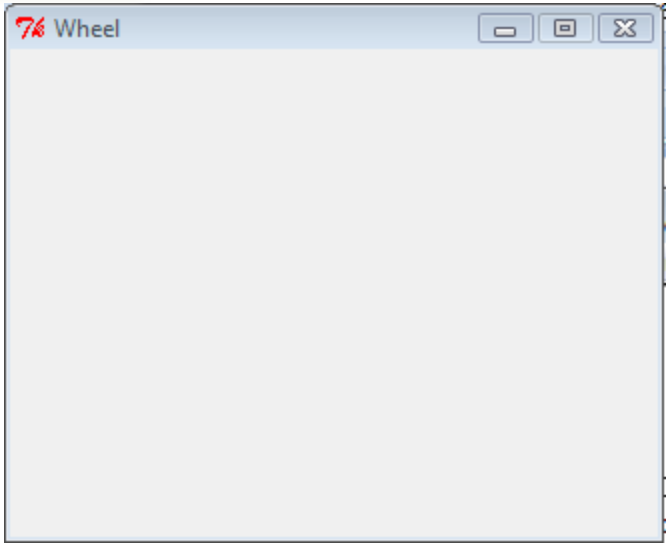
Wheel Class Definition

```
……...

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()
```

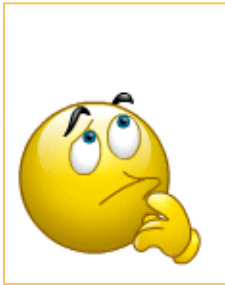

71

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

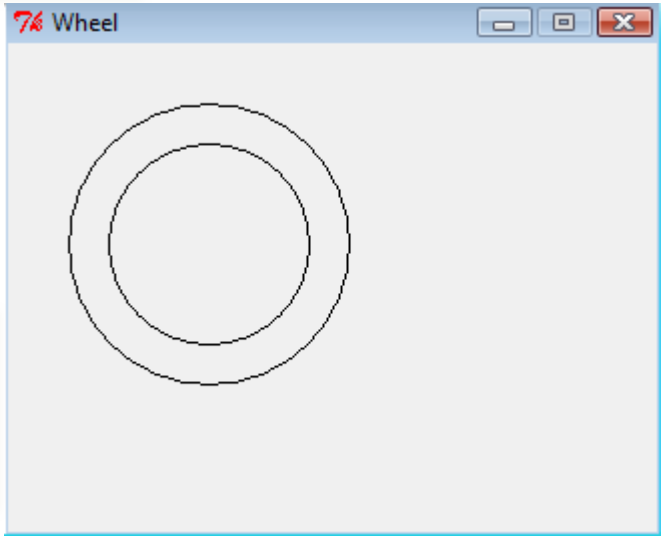


```
win = GraphWin('Wheel', 320, 240)
w = Wheel(Point(100, 100), 50, 70)
w.draw(win)
w.set_color('gray', 'black')
w.undraw()
win.mainloop()
```

What happened to the mysterious self parameter?

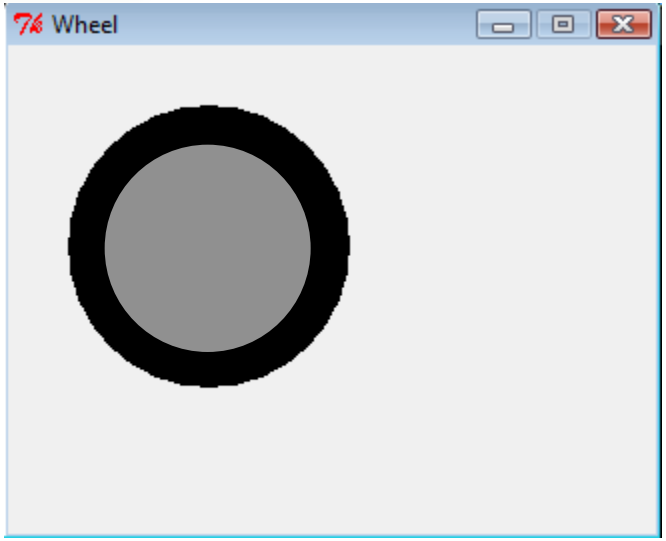
self = w

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



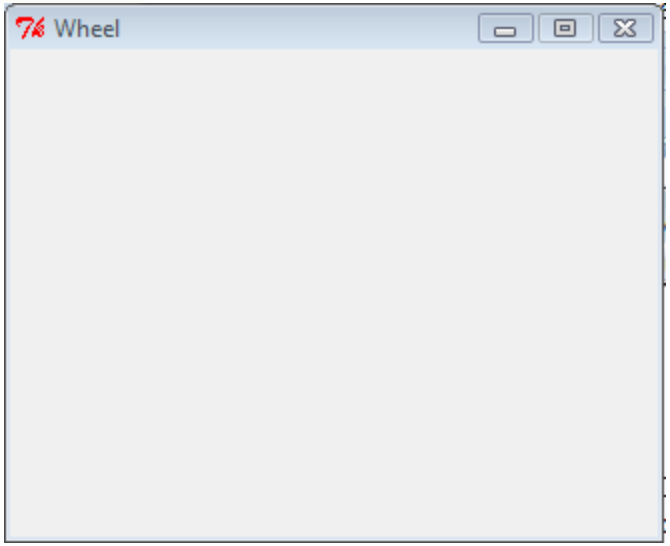
Using our Wheel class

```
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()
```



Exceptions

Exceptional Situations

```
def calculate_infinity():
    infinity = 3/0
    return infinity
```

Exceptional Situations

```
def calculate_infinity():
    infinity = 3/0
    return infinity
```

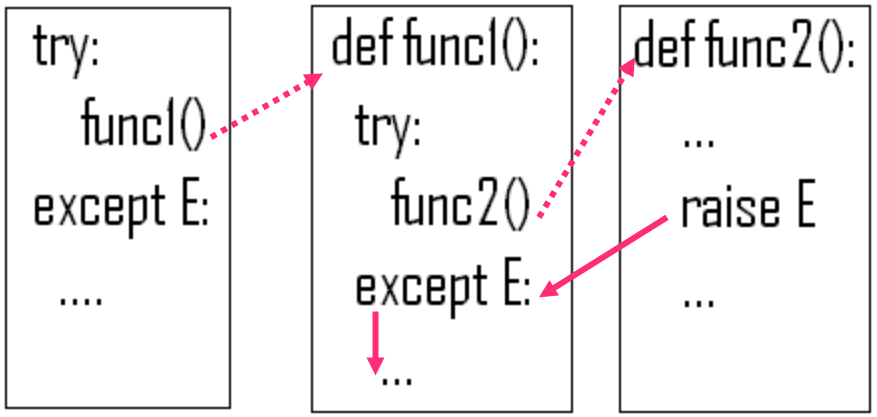
Exception Terminology

- **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

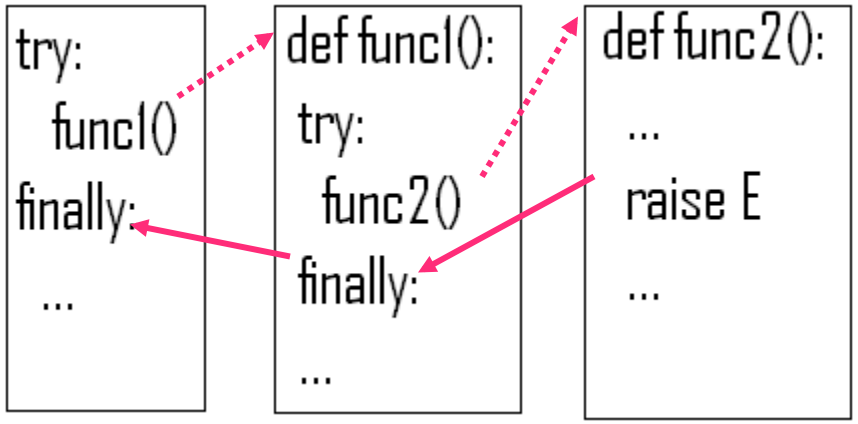
```
try:
    code to try

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.

Exception Idioms

- All errors are exceptions, but not all exceptions are errors. It could be signals or warnings

```
>>>while True:
        try:
            line=raw_input()
        except EOFError:
            break
        else:
            # process next line
```

- Functions signal conditions with *raise* (to distinguish success or failure)

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
```

User Defined Exceptions

```
raise MyError('oops!')
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

Regular Expressions

Regular Expressions

- A powerful string manipulation tool
- Can specify the rules for which you can match strings
- REs are compiled into a series of bytecodes which are then executed by a regex engine
- All modern languages have similar library packages for regular expressions

Examples of Regular Expressions

- "What text is embedded in the <H3> tag?"
- Validating email addresses.
- Parsing huge files of structured text to extract information

Regular Expressions

- Use regular expressions to:
  - Search a string (`search` and `match`)
  - Replace parts of a string (`sub`)
  - Break strings into smaller pieces (`split`)

Regular Expression Python Syntax

- Most characters match themselves

  The regular expression “test” matches the string `'test'`, and only that string

- [x] matches any *one* of a list of characters

  “[abc]” matches `'a'`,`'b'`,or `'c'`

- [^x] matches any *one* character that is not included in *x*

  “[^abc]” matches any single character *except* `'a'`,`'b'`,or `'c'`

Anchors (Position Characters)

- Anchors allow you to designate where a match can occur
  - ^ → match to beginning of String
    - Example:
      - Pattern: "^[Aa] [Rr]ose"
      - "A Rose is a rose is a rose."
  - $ → match at end of String
    - Example
      - Pattern: "rose$"
      - "A Rose is a rose is a rose"

Anchors (Position Characters)

- \b matches at word boundary:
  - Pattern "\brose" matches "rose" "rosemary", but not "primrose"

Repetition Operators

- Repetition operators allow us to denote that a (sub)pattern may repeat
  - * → zero or more repetitions
    - Example: "0*\d" matches "05" "5" "0006"
  - + → One or more repetitions
    - Example: "de+r" matches "deer" "deeer" "der" not "debr"
  - ? → exactly zero or 1 occurrence
    - Example "de[ae]?r" matches "der" "deer" "dear" not "debr" "deeer"
  - {m, n} → at least *m* occurrences and at most *n* occurances

    “a{2,3}$” matches ”aa” or ”aaa" but not ”a" or ”aaaa”

Grouping

- Just like math expressions you can group subpatterns using ( )
  - Pattern "(word)+" matches "word" "wordword" "wordwordword" not "" "wordd"

Example: Valid Email Address

- aiti@mit.edu
  - one or more word characters
  - followed by '@'
  - followed by word characters that has to have at least one '.' somewhere
    - (since '.' is an operator in a RE, we need to escape it)

Example: Valid Email Address

(\w)+@\w+(\.\w)+

Escaping

- If you want one of the RE reserved characters to appear in your pattern you must escape it:
  - \. → literal . in pattern
  - * → literal * in pattern
  - \ { } + ( ) are the others you must escape

Alternation

- | denotes logical OR operation
- Examples:
  - Pattern "soda|juice" matches "soda" "juice" "soda water", not "water"
  - "\w+@[\w\.]*\.(net|gov|edu)"
    - Good or bad RE for emails?
- | has lowest precedence (applied last)
  - Use ( ) to avoid confusion

Pairs

- “\d” matches any digit; “\D” matches any non-digit
- “\s” matches any whitespace character; “\S” matches any non-whitespace character
- “\w” matches any alphanumeric character; “\W” matches any non-alphanumeric character
- “\b” matches a word boundary; “\B” matches position that is not a word boundary
  (A word boundary is a position that changes from a word character to a non-word character, or vice versa.)

Search and Match

- The two basic functions are **re.search** and **re.match**
  - Search looks for a pattern anywhere in a string
  - Match looks for a match starting at the beginning
- Both return **None** if the pattern is not found (logical false) and a “match object” if it is

```
>>> pattern= "a*b"
>>> import re
>>> re.search(pattern,"fooaaabcde")
<_sre.SRE_Match object at 0x809c0>
>>> re.match(pattern,"fooaaabcde")
>>>
```

Match object

An instance of the match class with the details of the match result

```
>>>pattern= "a*b"
>>>r1 = re.search(pattern,"fooaaabcde")
>>>r1.group()  # group returns string matched
'aaab'
>>>r1.start()  # index of the match start
3
>>>r1.end()    # index of the match end
7
>>>r1.span()   # tuple of (start, end)
(3, 7)
```

What got matched?

- We can ‘label’ the groups as well…
  (?P<*label*>*regular expression*)

```
>>> pattern="(?P<name>\w+)@(?P<host>(\w+\.)+(com|
  org|net|edu))"
>>> r2 = re.match(pattern,"aiti@mit.edu")
>>> r2.group('name')
'aiti'
>>> r2.group('host')
'mit.edu'
```

Compiling regular expressions

- If you plan to use a re pattern more than once, compile it to a re object
- Python produces a special data structure that speeds up matching

```
>>> pattern="(?P<name>\w+)@(?P<host>(\w+\.)+(com|org|
   net|edu))"
>>> compiled= re.compile(pattern)
>>> compiled
<_sre.SRE_Pattern object at 0x2d9c0>
>>> r3 = compiled.search("aiti@mit.edu")
>>> r3
<_sre.SRE_Match object at 0x895a0>
>>> r3.group()
'aiti@mit.edu'
```

Substitution

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

Examples

```
>>> re.sub('\d', 'x', 'a_b - 12')
'a_b - xx'
>>> re.sub('\D', 'x', 'a_b - 12')
'xxxxxxx12'
>>> re.sub('\s', 'x', 'a_b - 12')
'a_bx-x12'
>>> re.sub('\S', 'x', 'a_b - 12')
'xxx x xx'
>>> re.sub('\w', 'x', 'a_b - 12')
'xxx - xx'
>>> re.sub('\W', 'x', 'a_b - 12')
'a_bxxx12'
```

How to be a Python Ninja!

Python Pow

- In encryption, we like to do (a^b)%c
- A, b, and c can be very large numbers.
- Ex: `(1234567890**9876543219)%33`
  - This is very slow. (wasn't done in 3 hours)
  - 650MB of ram, processor maxed out.
- Better way: `pow(1234567890,9876543219,33)`
  - At least 1800x faster. (6.14 seconds)
  - The answer is 24.

Reading a text file

- Easy in python:

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

Timing your code

```
import Timer
t = timeit.Timer(“8**2”)
print t.timeit()
```

- If you want to time something longer, use the timer to call a method.

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”
```

Sets

- Sets don’t have duplicate values.
- If you only want unique values in a list, you can create a set from it:

```
Print set([1,1,2,2,2,3,3,3,3,4])
```

- Output: set([1,2,3])

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]
```

If any, if all

- `numbers = [1,2,3,4,5,6,7]`
- `If any(num for num in numbers)>6`
  - True if any number is greater than 6
- `If all(num for num in numbers)>6`
  - True only if all numbers are greater than 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]]
```

Generators

- Generators have the same syntax as list comprehensions, but use parenthesis instead of square brackets
- These are faster than list comprehensions and use much less memory, but can't store your data.
- Computes one value at a time.

Generators

- List comprehension
  - sum([a^b for a in range(1000) for b in range (1000)])
  - The complete list comprehension is created first, stored in memory, and summed after completion.
  - 25 seconds, >600MB ram
- Generator
  - sum(a^b for a in range(1000) for b in range (1000))
  - Values are added to the sum one at a time
  - 23 seconds, <0.5MB ram

Lambda functions

- A function that is created at runtime.
- Always returns something (but doesn't include a return statement)
- Convenient for passing as an argument
- Ex:

```
f = lambda x:x**2
```

- Takes x as input and returns x^2

Filter Function

- Syntax: filter(function, list)
- Ex:

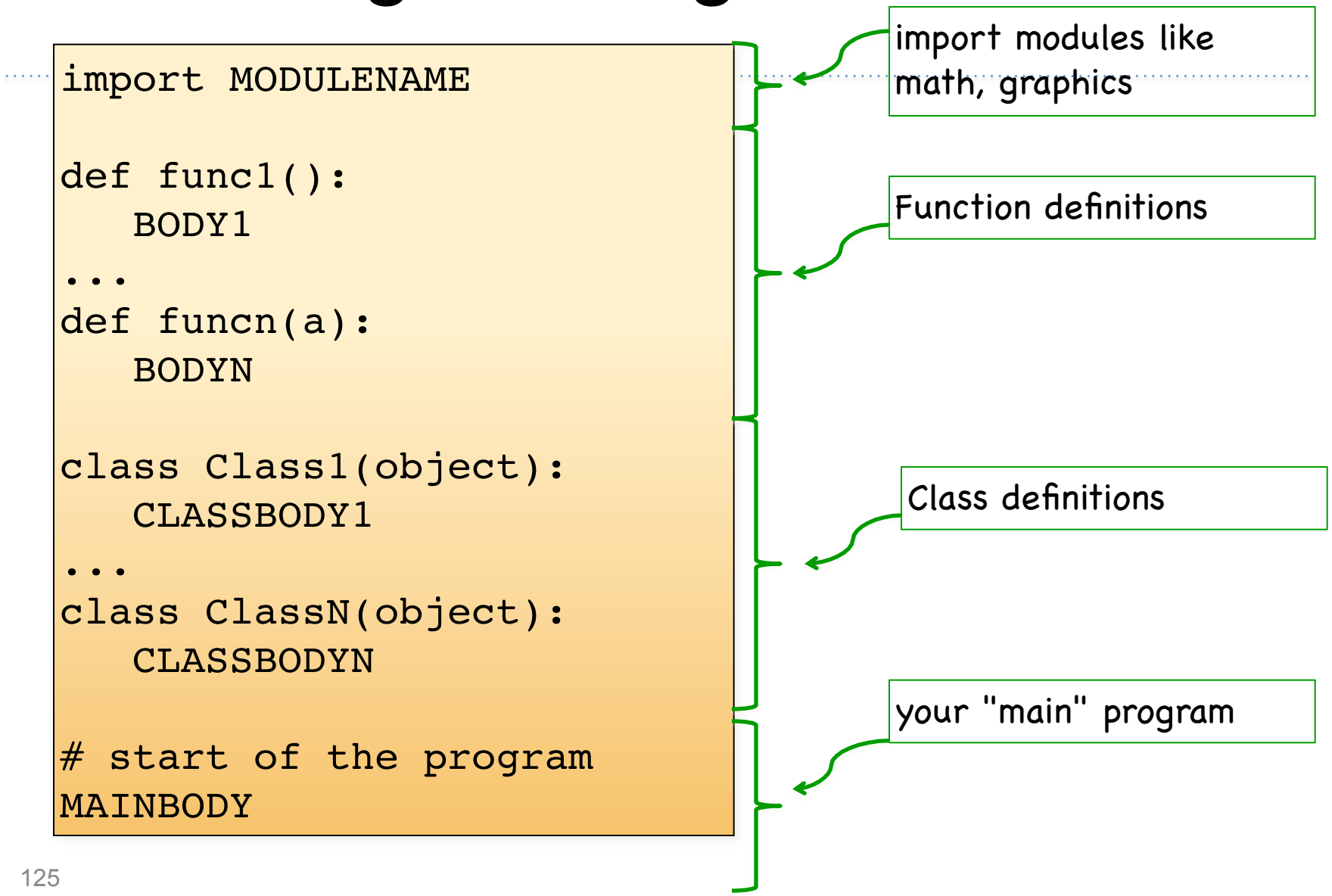
```
numbers = [1,2,3,4,5,6,7]
print filter(lambda x: x<4, numbers)
```

Output:

```
[1,2,3]
```

Program Organization

```
import MODULENAME

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

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

# start of the program
MAINBODY
```



Lab 2

- OOP
  - Animate a Wheel
  - Implement Car class and animate it
  - Implement Polynomial class to perform algebraic operations on polynomials
- Regular Expressions
  - Validate an email address
  - Extract a data item from a Web page
- Be a Python Ninja!
  - Python one liners!
  - Polyomino: determine the number of common proteins