Introduction to Python Programming

General Information

- Unlike C/C++ or Java, Python statements do not end in a semicolon
- Whitespace is meaningful in Python: especially indentation and placement of newlines.
- In Python, indentation is the way you indicate the scope of a conditional, function, etc.
- Look, no braces!
- Python is interpretive, meaning you don’t have to write programs.
- You can just enter statements into the Python environment and they’ll execute

The Python Shell

- Because Python is interpretive, you can do simple things with the shell
- At the prompt, type `python`
- You should have a >>> prompt
- Type in:

```
print(“hello, world”)
```

- You have written your first Python program
- Save it as as HelloWorld.py
- You **must** provide the .py extension
- `Type python HelloWorld.py`
- **Note that Linux is case sensitive**

print

- `print` : Produces text output on the console.
- Syntax:

```
print "Message"
print Expression
print Item1, Item2, ..., ItemN
```

- Examples:

```
print "Hello, world!"
age = 45
print "You have", 65 - age, "years until retirement"
```

```
>>> Hello, world!
 You have 20 years until retirement
```

```
print  "%s xyz %d"%("abc", 34)
```

```
>>> abc xyz 34
```

input

input : Reads a number from user input.

You can assign (store) the result of `input` into a variable.

Example:

```
age = 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
```

Python and type

Python determines the data types in a program automatically. “Dynamic Typing”

- That is, you don’t declare variables to be a specific type
- A variable has the type that corresponds to the value you assign to it

But Python’s not casual about types, it enforces them after it figures them out. “Strong Typing”

So, for example, you can’t just append an integer to a string. You must first convert the integer to a string itself.

```
>>>
x = “the answer is ”   # Decides x is string.
y = 23                 # Decides y is integer.
print x + y    # Python will complain about this.
type(x)
type(y)
```

Variables

- As in every language, a variable is the name of a memory location

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.

  `bob   Bob   _bob   _2_bob_   bob_2   BoB`

- There are some reserved words which can not be used as variables:

  `and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while`

Variables

- No need to declare
- Need to assign (initialize)
    - use of uninitialized variable raises exception
- Not typed

```
if friendly: greeting = "hello world"
else: greeting = 12**2
print greeting
```

- If you try to access a name before it's been properly created (by placing it on the left side of an assignment), you'll get an `NameError` error.

Variables

- You can also assign to multiple names at the same time.

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
```

Variables

Assignment manipulates references

x = y **does not make a copy** of y

x = y makes x **reference** the object y references

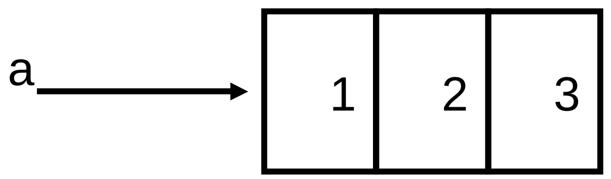
Very useful; but beware!

Example:

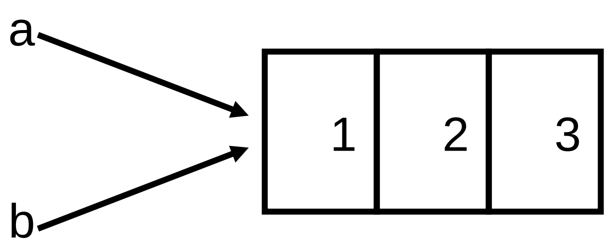
```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> print b
[1, 2, 3, 4]
```

Changing a shared list

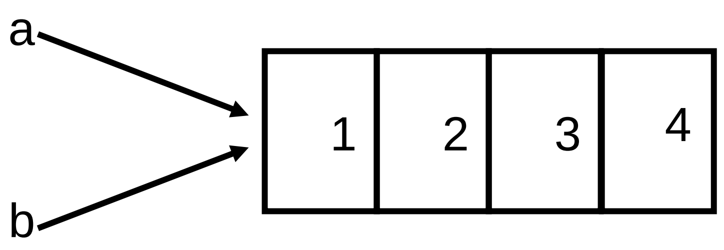
```
a = [1, 2, 3]
```



```
b = a
```



```
a.append(4)
```

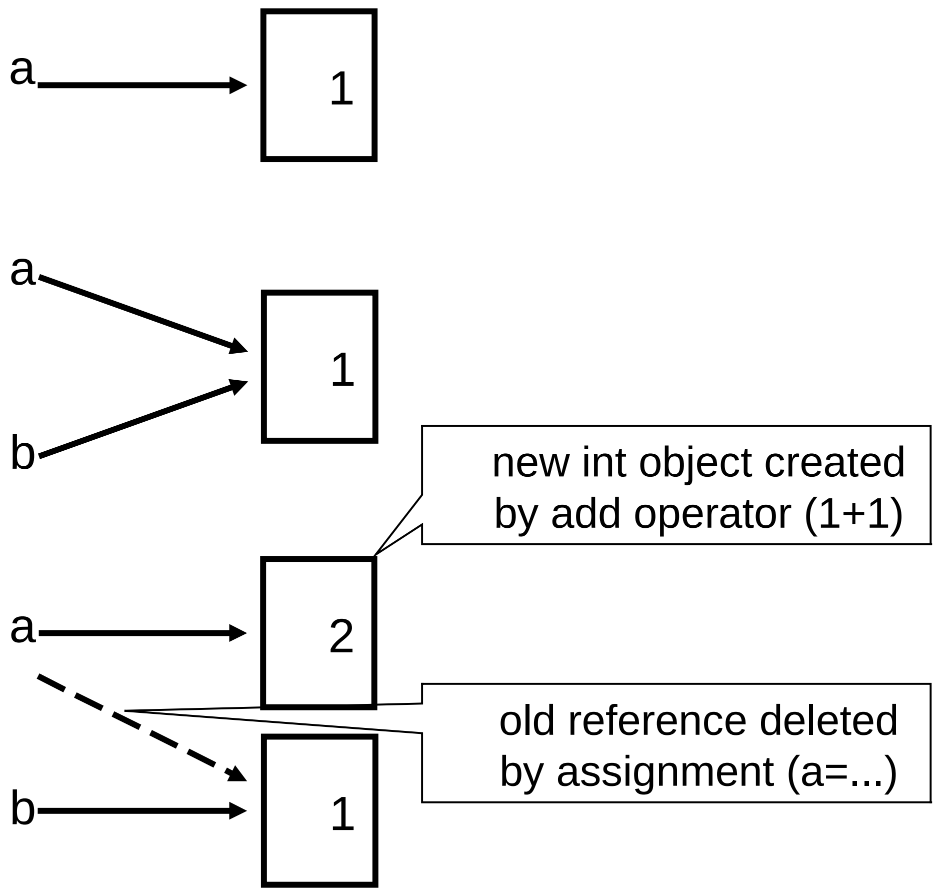


Changing an integer

a = 1

b = a

a = a+1



Comments

- All code must contain comments that describe what it does
- In Python, lines beginning with a # sign are comment lines

```
# This entire line is a comment
x=5   # Set up loop counter
```

Operators

- Arithmetic operators we will use:

  `+ - * /` addition, subtraction/negation, multiplication, division

`%`	modulus, a.k.a. remainder
`**`	exponentiation

- **precedence**: Order in which operations are computed.
  - `* / % **` have a higher precedence than `+ -`

    `1 + 3 * 4` is `13`

  - Parentheses can be used to force a certain order of evaluation.

    `(1 + 3) * 4` is `16`

Expressions

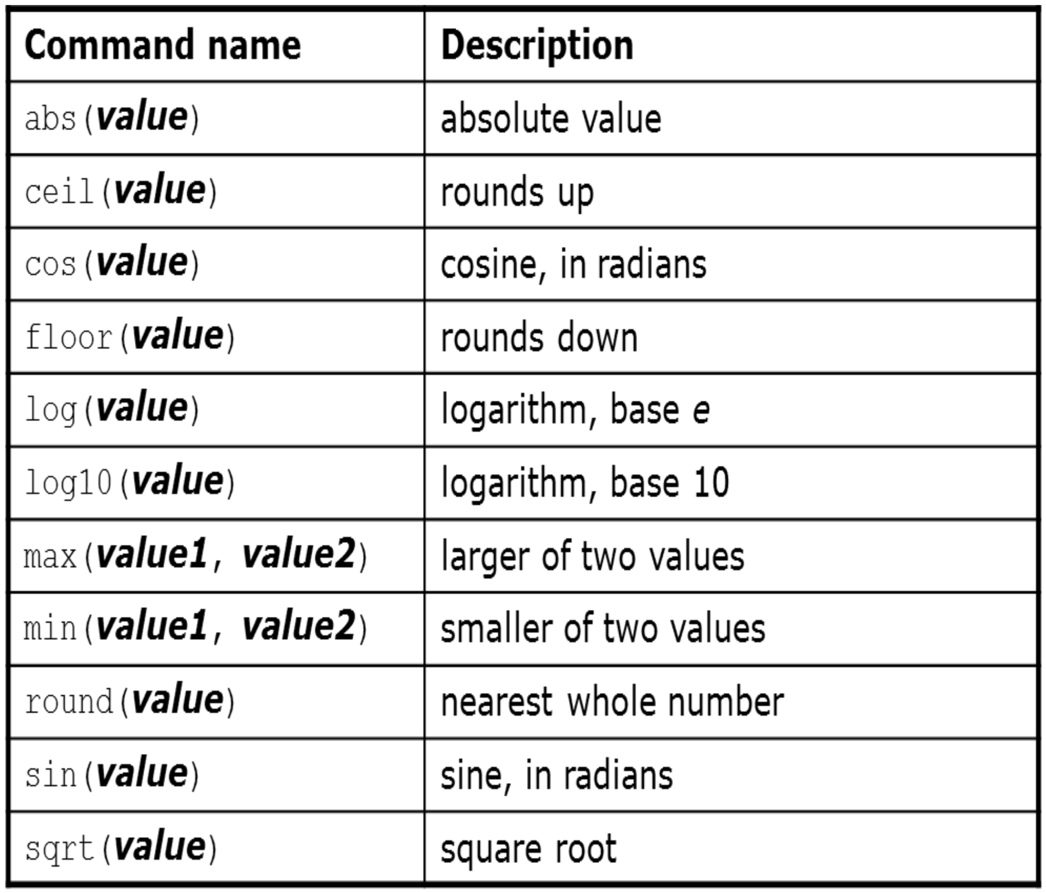
- When integers and reals are mixed, the result is a real number.
  - Example: `1 / 2.0` is `0.5`

  - The conversion occurs on a per-operator basis.
  - Integer division truncates :-(

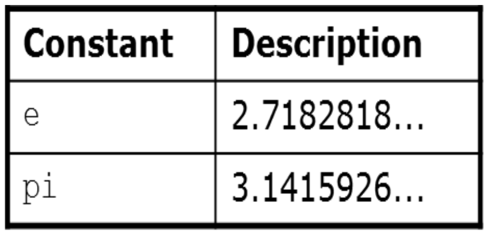
```
7 / 3 * 1.2 + 3 / 2
  2   * 1.2 + 3 / 2
    2.4     + 3 / 2
    2.4     +   1
         3.4
```

Math Functions

- Use this at the top of your program: `from math import *`



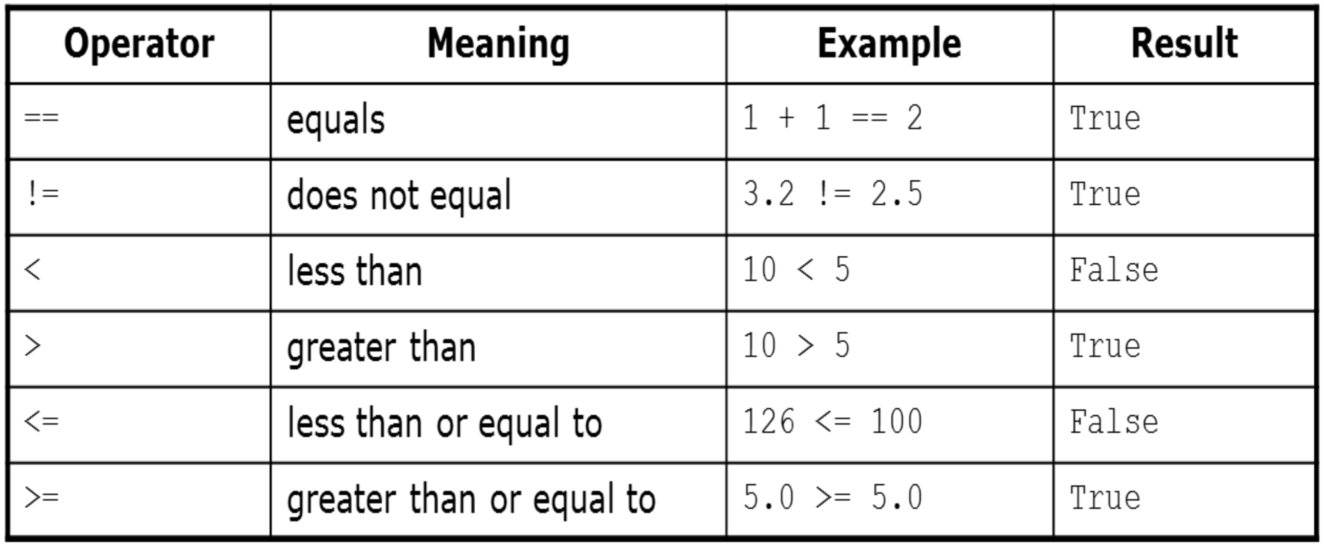
Command name	Description
abs(***value***)	absolute value
ceil(***value***)	rounds up
cos(***value***)	cosine, in radians
floor(***value***)	rounds down
log(***value***)	logarithm, base *e*
log10(***value***)	logarithm, base 10
max(***value1***, ***value2***)	larger of two values
min(***value1***, ***value2***)	smaller of two values
round(***value***)	nearest whole number
sin(***value***)	sine, in radians
sqrt(***value***)	square root



Constant	Description
e	2.7182818...
pi	3.1415926...

Relational Operators

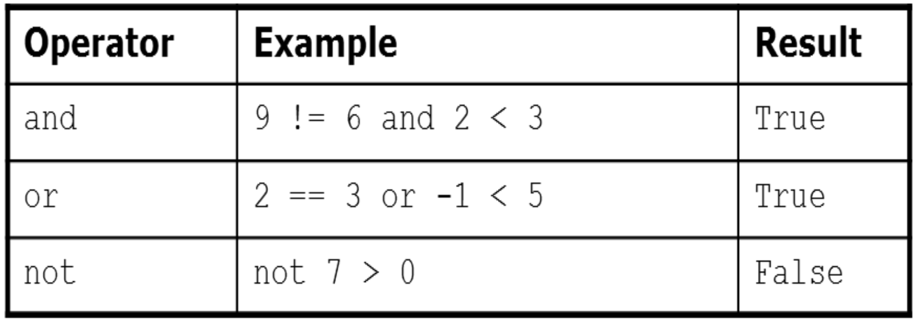
- Many logical expressions use *relational operators*:



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

Logical Operators

- These operators return true or false



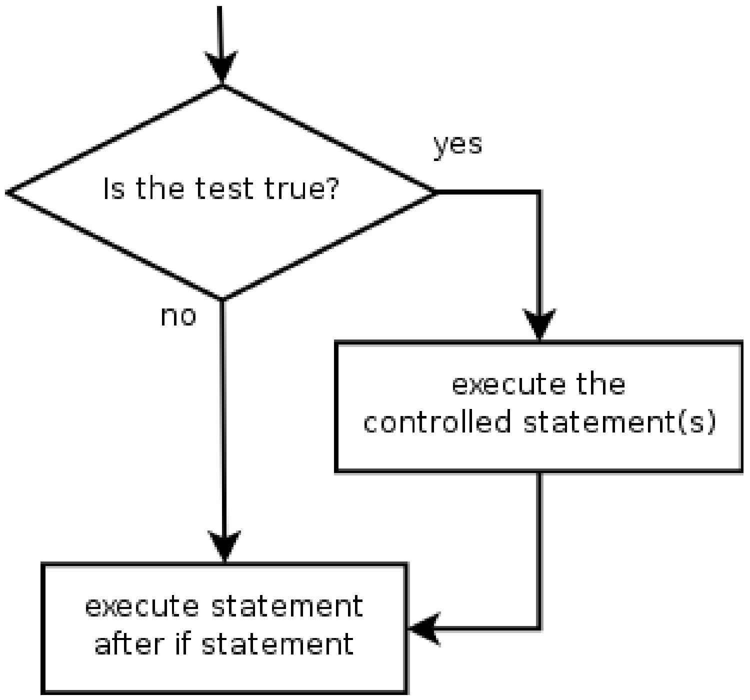
Operator	Example	Result
`and`	`9 != 6 and 2 < 3`	`True`
`or`	`2 == 3 or -1 < 5`	`True`
`not`	`not 7 > 0`	`False`

The `if` Statement

- Syntax:

```
if <condition>:
   <statements>
```

```
x = 5
if x > 4:
   print(“x is greater than 4”)
print(“This is not in the scope of the if”)
```

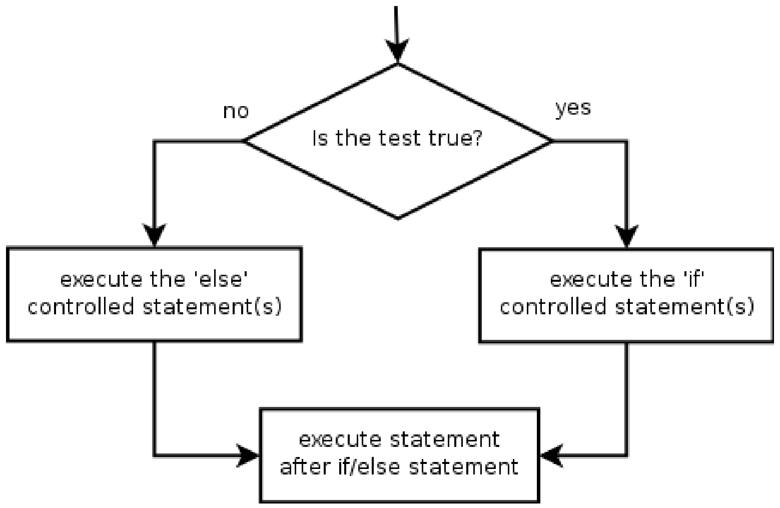

Is the test true?
yes
no
execute the controlled statement(s)
execute statement after if statement

The if/else Statement

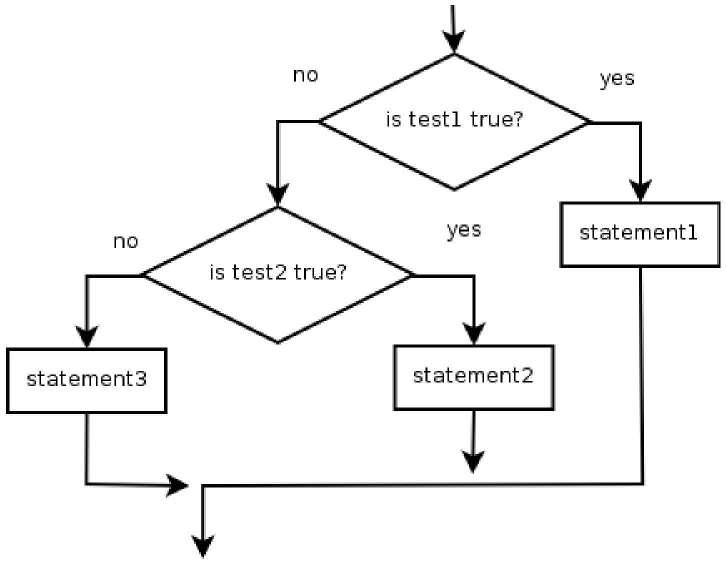
```
if <condition>:
    <statements>
else:
    <statements>
```

```
if <condition>:
    statements
[elif condition:

  statements] ...
else:
    statements
```



no
yes
Is the test true?
execute the 'else' controlled statement(s)
execute the 'if' controlled statement(s)
execute statement after if/else statement



The for Loop

- This is similar to what you're used to from C or Java, but not the same
- Syntax:

```
for variableName in groupOfValues:
    <statements>
```

- variableName gives a name to each value, so you can refer to it in the statements.
- groupOfValues can be a range of integers, specified with the range function.

Example:

```
for x in range(1, 6):
    print x, "squared is", x * x
```

Output:

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
```

Range

- The range function specifies a range of integers:

`range(start, stop)` - the integers between start (inclusive) and stop (exclusive)

- It can also accept a third value specifying the change between values.

`range(start, stop, step)` - the integers between start (inclusive) and stop (exclusive) by step

Example:

```
for x in range(5, 0, -1):
    print x
print "Blastoff!"
```

Output:

```
5
4
3
2
1
Blastoff!
```

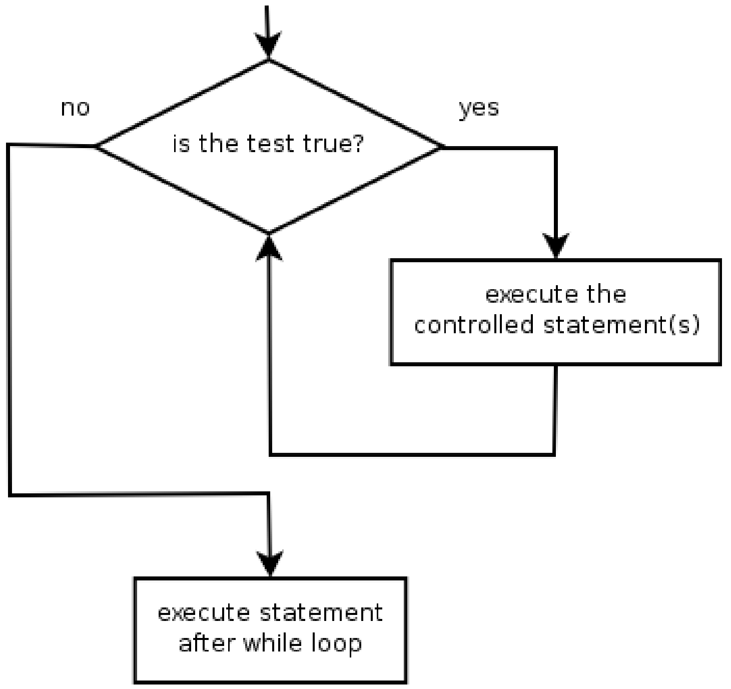
The while Loop

- Executes a group of statements as long as a condition is True.
- Good for indefinite loops (repeat an unknown number of times)
- Syntax:

```
while <condition>:
    <statements>
```

- Example:

```
number = 1
while number < 200:
    print number,
    number = number * 2
```



no
yes
is the test true?
execute the controlled statement(s)
execute statement after while loop

Grouping Indentation

In Python:

```
for i in range(20):
    if i%3 == 0:
        print i
        if i%5 == 0:
            print "Bingo!"
    print "---"
```

In C:

```
for (i = 0; i < 20; i++)
{
    if (i%3 == 0) {
        printf("%d\n", i);
        if (i%5 == 0) {
            printf("Bingo!\n"); }
      }
      printf("---\n");
}
```

```
0
Bingo!
---
---
---
3
---
---
---
6
---
---
---
9
---
---
---
12
---
---
---
15
Bingo!
---
---
---
18
---
---
```

Strings

- String: A sequence of text characters in a program.
- Strings start and end with quotation mark " or apostrophe ' characters.
- Examples:
  "hello"
  "This is a string"
  "This, too, is a string.   It can be very long!"
- A string may not span across multiple lines or contain a " character.
  "This is not
  a legal String."

  "This is not a "legal" String either."

Strings-operators

"hello"+"world"	"helloworld"	# concatenation
"hello"*3	"hellohellohello"	# repetition
"hello"[0]	"h"	# indexing
"hello"[-1]	"o"	# (from end)
"hello"[1:4]	"ell"	# slicing
len("hello")	5	# size
"hello" < "jello"	1	# comparison
"e" in "hello"	1	# search
escapes: \n \t etc.		

'single quotes' "double quotes" """triple quotes""" r"raw strings"

String Functions

- `len(`***`string`***`)` - number of characters in a string
- `str.lower(`***`string`***`)` - lowercase version of a string
- `str.upper(`***`string`***`)` - uppercase version of a string
- `str.isalpha(`***`string`***`)` - True if the string has only alpha chars
- Many others: split, replace, find, format, etc.
  - >>> "hello..{0}".format("world")
  - 'hello..world'

Other Built-in Types

- tuples, lists, sets, and dictionaries
- They all allow you to group more than one item of data together under one name
- You can also search them

Tuples

- Unchanging Sequences of Data
- Enclosed in parentheses:

```
tuple1 = (“This”, “is”, “a”, “tuple”)
print(tuple1)
```

- This prints the tuple exactly as shown

```
Print(tuple1[1])
```

- Prints “is” (without the quotes)

Tuples

```
key = (lastname, firstname)
point = x, y, z     # parentheses optional
x, y, z = point   # unpack
lastname = key[0]
singleton = (1,)  # trailing comma!!!
empty = ()                    # parentheses!
```

<u>tuples vs. lists</u>->tuples are immutable

Lists

- Changeable sequences of data

Lists are created by using square brackets:

```
breakfast = [ "coffee", "tea", "toast",
"egg" ]
```

- You can add to a list:

```
breakfast.append("waffles")
breakfast.extend(["cereal", "juice"])
```

- Same operators as for strings

a+b, a*3, a[0], a[-1], a[1:], len(a)

Lists

```
>>> a = range(5)            # [0,1,2,3,4]
>>> a.append(5)             # [0,1,2,3,4,5]
>>> a.pop()                 # [0,1,2,3,4]
5
>>> a.insert(0, 42)         # [42,0,1,2,3,4]
>>> a.pop(0)                # [0,1,2,3,4]
42
>>> a.reverse()             # [4,3,2,1,0]
>>> a.sort()                # [0,1,2,3,4]
```

Dictionaries

- Groupings of Data Indexed by Name
- Dictionaries are created using braces

```
sales = {}
sales["January"] = 10000
sales["February"] = 17000
sales["March"] = 16500
```

- The keys method of a dictionary gets you all of the keys as a list
- Hash tables, "associative arrays"

```
{"January":10000, "February":17000,"March":16500}
```

- Lookup :

```
sales["March"]
```

Dictionaries

- Keys, values, items:

sales.keys() -> [`“January,“February”,“March”`]

sales.values() -> [`10000,17000,16500`]

sales.items() ->

[(`“January”,10000),(“February”,17000),(“March”:16500)`]

- Presence check:

  sales.has_key("January") ->
  `1`

  sales.has_key("spam") ->
  `0`

- Values of any type; keys almost any
  - {"name":"Guido", "age":43, ("hello","world"):1, 42:"yes", "flag": ["red","white","blue"]}

Dictionaries

- Keys must be **immutable:**
  - numbers, strings, tuples of immutables
    - these cannot be changed after creation
  - reason is *hashing* (fast lookup technique)
  - **not** lists or other dictionaries
    - these types of objects can be changed "in place"
  - no restrictions on values
- Keys will be listed in **arbitrary order**
  - again, because of hashing

Sets

- Sets are similar to dictionaries in Python, except that they consist of only keys with no associated values.
- Essentially, they are a collection of data with no duplicates.
- They are very useful when it comes to removing duplicate data from data collections.
- Can do set operations – union(|),intersection(&), difference(-) and symmetrical difference (^), x in set

```
>>> s = {1,2,3,2,1}
>>> s
set([1, 2, 3])
>>> s = set([1,2,2,3,3,1,1])
>>> s
set([1, 2, 3])
```

Writing Functions

- Define a function:

```
def <function name>(<parameter list>):
    """documentation"""      # optional doc string
    # The code would go here…
```

The function body is indented one level:

```
def computeSquare(x):
    return x * x
#anything here is not the part of the function
```

Example Functions

```
def gcd(a, b):
    "greatest common divisor"
    while a != 0:
        a, b = b%a, a    # parallel assignment
    return b

>>> gcd.__doc__
'greatest common divisor'
>>> gcd(12, 20)
4
```

Error Handling-try/except

- Use try/except blocks, similar to try/catch:

```
fridge_contents = {"egg":8, "mushroom":20,
"pepper":3, "cheese":2,
"tomato":4, "milk":13}
try:
    if fridge_contents["orange juice"] > 3:
        print("Sure, let’s have some juice!")
except KeyError:
    print("Awww, there is no orange juice.")
```

Error Handling:raise

- raise IndexError
- raise IndexError("k out of range")
- raise IndexError, "k out of range"
- Last caught exception info:
  - sys.exc_info() == (exc_type, exc_value, exc_traceback)
- Last uncaught exception (traceback printed):
  - sys.last_type, sys.last_value, sys.last_traceback

```
import sys
try:
    1/0
except:      # catch everything
  print "Oops:", sys.exc_info()
  raise      # reraise
```

Python File I/O:open

- You can read and write text files in Python much as you can in other languages, and with a similar syntax.
- The mode indicates, how the file is going to be opened "r" for reading, "w" for writing and "a" for a appending.
- To open a file for reading:

```
try:
    file = open("test.txt", "r")
    #do something with the opened file
except IOError as err:
    print("could not open file: " + str(err))
```

Python File I/O

- The read functions contains different methods, read(),readline() and readlines()
  - read() #return one big string
  - readline #return one line at a time
  - readlines #returns a list of lines
- To read from a file:

```
while 1:
  line = file.readline()
  print line
  if len(line) == 0:
    break
```

Python File I/O:read

- You can also read all lines from a file into a set, then iterate over the set:

```
file = open("test.txt", "r")
lines = file.readlines()
for line in lines:
    print(line)
```

```
file = open("test.txt", "r")
print file.read()
```

Python File I/O:write

- This method writes a sequence of strings to the file.
  - write () #Used to write a fixed sequence of characters to a file
  - writelines() #writelines can write a list of strings.
- Writing to a text file

```
file=open("test.txt","w")
file.write("This is how you create a new text file")
file.close()
```

Python File I/O:close

- *Try the following code with nonexisting file!*

Explicit use

(what is the problem here?)

```
try:

    file = open("test.txt", "r")
    print file.read()

except IOError as err:

    print("could not open file: "
+ str(err))

finally:

    file.close()
```

```
try:

    file = open("test.txt", "r")
    print file.read()
    file.close()

except IOError as err:

    print("could not open file: "
+ str(err))
```

Python File I/O:close

- *Try the following code with nonexisting file!*

Use of additional try

```
try:
  file = open("test.txt", "r")
  try:
   # Do something with file
   print file.read()
  finally:
   file.close()
except IOError as err:
  print("could not open file: " + str(err))
```

Python File I/O:close

- *Try the following code with nonexisting file!*

Use of *with* to close the file

(solution: file.__enter__(), file.__exit__() used by *with*)

```
try:
 with open("test.txt","r") as file:
  print file.read()
except IOError as err:
 print("could not open file: ",str(err))
```

Graphics

from graphics import *

```
g=GraphWin()
c= Circle(p, 50)
c.setOutline('blue')
c.setFill("red")
c.draw(g)
g.close()
```

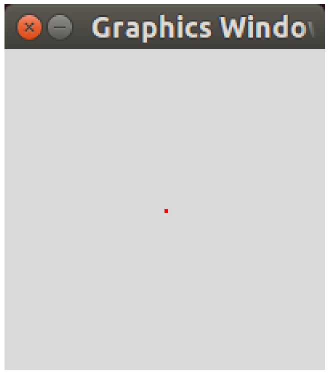
```
g=GraphWin()
o= Oval (Point(50,50), Point(150,100))
o.setFill("red")
o.draw(g)
g.close()
```

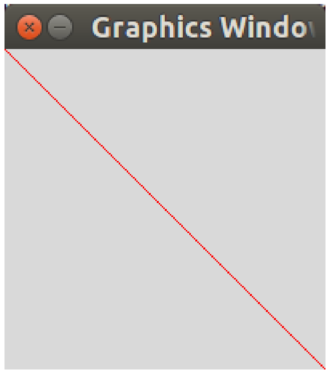
Graphics

from graphics import *

```
g=GraphWin()
p = Point(100,100)
p.setFill("red")
p.draw(g)
g.close()
```

```
g=GraphWin()
l = Line(Point(0,0),Point(200,200))
l.setFill("red")
l.draw(g)
g.close()
```



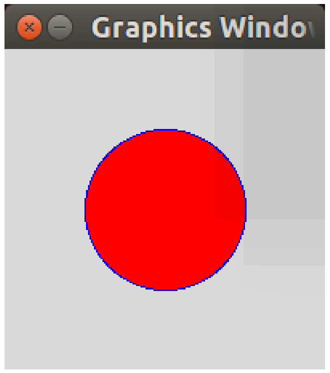


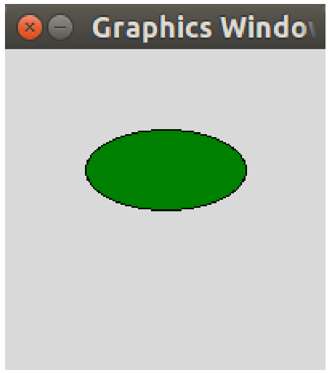
Graphics

from graphics import *

```
g=GraphWin()
c= Circle(Point(100,100), 50)
c.setOutline('blue')
c.setFill("red")
c.draw(g)
g.close()
```

```
g=GraphWin()
o= Oval (Point(50,50), Point(150,100))
o.setFill("green")
o.draw(g)
g.close()
```

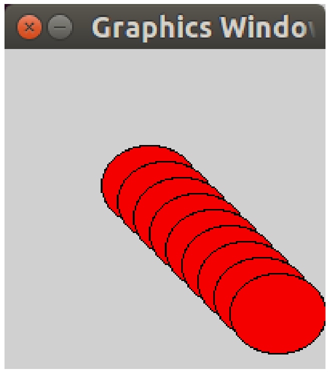




Graphics

from graphics import *

```
g=GraphWin()
for I  in range(1,10):
   o= Oval (Point(50 + I*10, 50 + I*10), Point(110 + I*10, 100 + I*10))
   o.setFill("red")
   o.draw(g)
g.close()
```



Animation

```
from graphics import *
import time

def main(color, x, y, max):
  #create a graphics window
  g = GraphWin('Back and Forth', 300, 300)

  # create a circle
  c = Circle(Point(0,0), 25)
  c.setFill(color)
  c.setOutline("black")
  c.draw(g)

  # animate: move down to right
  for i in range(max):
    c.move(x, y)
    time.sleep(.05)

  # animate; move up to left
  for i in range(max):
    c.move(-x, -y)
    time.sleep(.05)

  g.close()

main('red', 5,5, 60)
```

```
colors  =
['red','yellow','green','blue','brown','pink']

#change the speed of movement and
color
for i in range(len(colors)):
   x, y, max = (i+1)*5, (i+1)*5,  60/(i+1)
   main(colors[i], x, y, max)
```