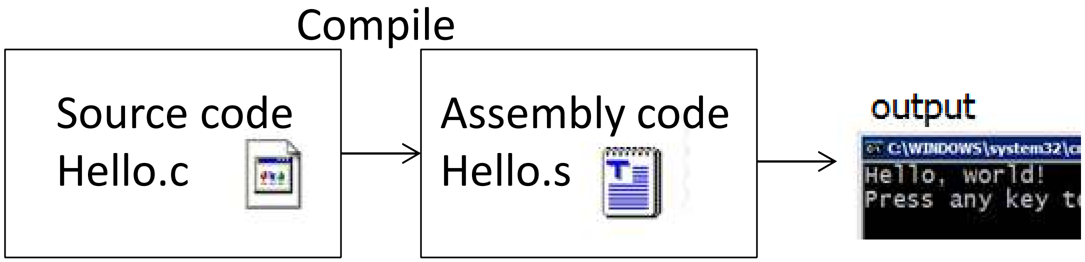
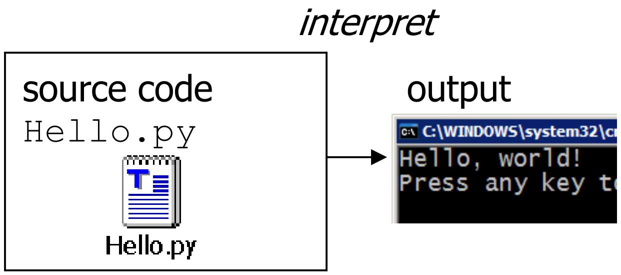
Compiling and interpreting

- Many languages require you to *compile* (translate) your program into a form that the machine understands.



```
Hello, world!
Press any key to
```

- Python is instead directly *interpreted* into machine instructions.



```
Hello, world!
Press any key to
```

Variables

Python

```
x = 5
y = x + 7
z = 3.14

name = “Rishi”


1 == 1 # => True
5 > 10 # => False

True and False # => False
not False # => True
```

Variables are not statically typed!

Java/C++

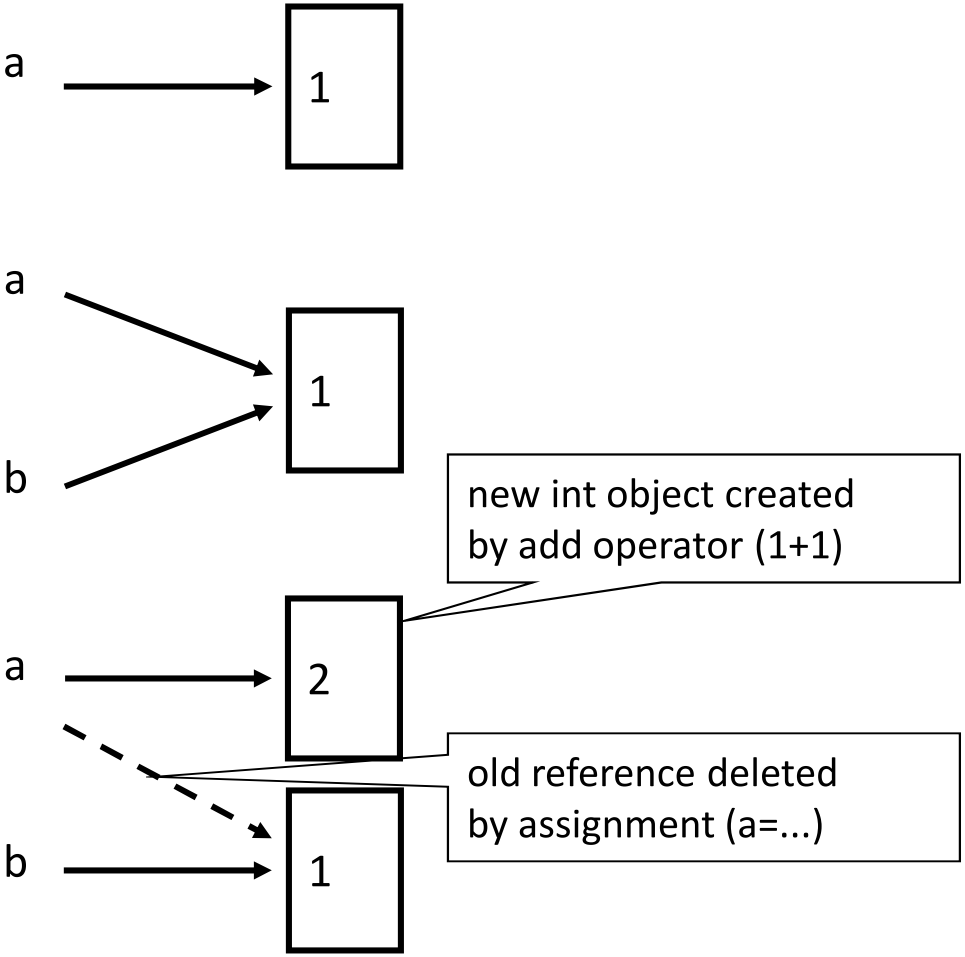
```
int x = 5;
int y = x + 7;
double z = 3.14;

String name = “Rishi”; // Java
string name(“Rishi”);  // C++

1 == 1 # => true
5 > 10 # => false

true && false # => false
!(false) # => true
```

Changing an Integer (Everything in python is a reference)

a = 1

b = a

a = a+1



new int object created by add operator (1+1)

old reference deleted by assignment (a=...)

Variable names

- Can contain letters, numbers, and underscores
- Must begin with a letter
- Cannot be one of the reserved Python keywords: and, as, 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, with, yield

Comments

- Anything after a # symbol is treated as a comment
- This is just like Perl

Operators

- + addition
- - subtraction
- / division
- ** exponentiation
- % modulus (remainder after division)
- Comparisons

Operators

-
  ```
  print (2*2)
  print (2**3)
  print (10%3)
  print (1.0/2.0)
  print (1/2)
  print(1//2)
  ```

  ```
  Output:
  4
  8
  1
  0.5
  0.5
   0
  ```

+= but not ++

- Python has incorporated operators like +=, but ++ (or --) do not work in Python

Type conversion

- int(), float(), str(), and bool() convert to integer, floating point, string, and boolean (True or False) types, respectively

-
```
print (int(3.1415926))
print (str(3.1415926))
print (bool(1))
print (bool(0))
```

- Output
```
 3
3.1415926
True
False
```

Strings

```
>>> smiles = "C(=N)(N)N.C(=O)(O)O"
>>> smiles[0]
'C'
>>> smiles[1]
'('
>>> smiles[-1]
'O'
>>> smiles[1:5]
'(=N)'
>>> smiles[10:-4]
'C(=O)'
```

Use “slice” notation to get a substring

String Methods: find, split

```
smiles = "C(=N)(N)N.C(=O)(O)O"
>>> smiles.find("(O)")
15
>>> smiles.find(".")
9
>>> smiles.find(".", 10)
-1
>>> smiles.split(".")
['C(=N)(N)N', 'C(=O)(O)O']
>>>
```

Use “find” to find the start of a substring.

Start looking at position 10.

Find returns -1 if it couldn’t find a match.

Split the string into parts with “.” as the delimiter

String operators: in, not in

```
if "Br" in “Brother”:
    print "contains brother“

email_address = “clin”
if "@" not in email_address:
    email_address += "@brandeis.edu“
```

Operators acting on strings

-
  ```
  >>> "Ni!"*3
  'Ni!Ni!Ni!'
  ```

-
  ```
  >>> "hello " + "world!"
  'hello world!'
  ```

More string basics

- Type conversion:

```
>>> int(“42”)
42
>>> str(20.4)
'20.4'
```

- Compare strings with the is-equal operator, == (like in C and C++):

```
>>> a = “hello”
>>> b = “hello”
>>> a == b
True
```

Unexpected things about strings

Strings are read only

```
>>> s = "andrew"
>>> s[0] = "A"
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> s = "A" + s[1:]
>>> s
'Andrew‘
```

“\” is for special characters

\n -> newline

\t -> tab

\\ -> backslash

...

But Windows uses backslash for directories!

```
filename = "M:\nickel_project\reactive.smi" # DANGER!
filename = "M:\\nickel_project\\reactive.smi" # Better!
filename = "M:/nickel_project/reactive.smi" # Usually works
```

Collection Data Types

- Lists
- Tuples
- Dictionaries

List

A compound data type:

```
[0]
[2.3, 4.5]
[5, "Hello", "there", 9.8]
[]
```

Use len() to get the length of a list

```
>>> names = [“Ben", “Chen", “Yaqin"]
>>> len(names)
3
```

Use [ ] to index items in the list

```
>>> names[0]
‘Ben'
>>> names[1]
‘Chen'
>>> names[2]
‘Yaqin'
>>> names[3]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> names[-1]
‘Yaqin'
>>> names[-2]
‘Chen'
>>> names[-3]
‘Ben'
```

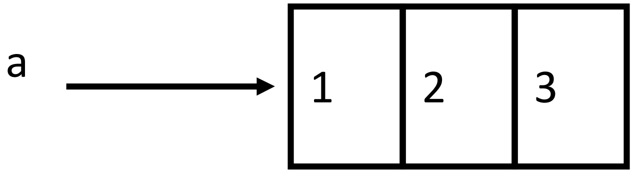
[0] is the first item.
[1] is the second item
…

Out of range values
raise an exception

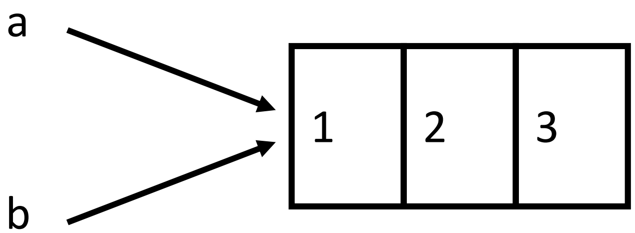
Negative values
go backwards from
the last element.

Changing a Shared List

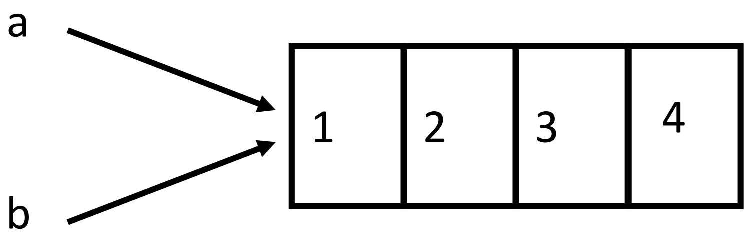
a = [1, 2, 3]



b = a



a.append(4)



Lists are mutable - some useful methods

```
>>> ids = ["9pti", "2plv", "1crn"]
>>> ids.append("1alm")
>>> ids
['9pti', '2plv', '1crn', '1alm']
```

append an element

```
>>> del ids[0]
>>> ids
['2plv', '1crn', '1alm']
```

Remove an element

```
>>> ids.sort()
>>> ids
['1alm', '1crn', '2plv']
```

sort by default order

```
>>> ids.reverse()
>>> ids
['2plv', '1crn', '1alm']
```

reverse the elements in a list

```
>>> ids.insert(0, "9pti")
>>> ids
['9pti', '2plv', '1crn', '1alm']
```

insert an element at some specified position.
(Slower than .append())

Lists Contains Object References

- Lists contains *object references*. Since lists are also objects, they can be nested

```
>>> a=[0,1,2]
>>> b=[a,3,4]
>>> print (b)
[[0, 1, 2], 3, 4]
>>> print (b[0][1])
1
>>> print (b[1][0])
... TypeError: 'int' object is unsubscriptable
```

Tuples: sort of an immutable list

```
>>> yellow = (255, 255, 0) # r, g, b
>>> yellow[0]
>>> yellow[1:]
(255, 0)
>>> yellow[0] = 0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

zipping lists together

```
>>> names
['ben', 'chen', 'yaqin']

>>> gender = [0, 0, 1]

>>> zip(names, gender)
[('ben', 0), ('chen', 0), ('yaqin', 1)]
```

Dictionaries

- Dictionaries are lookup tables.
- They map from a “key” to a “value”.

```
symbol_to_name = {
        "H": "hydrogen",
        "He": "helium",
        "Li": "lithium",
        "C": "carbon",
        "O": "oxygen",
        "N": "nitrogen"
}
```

- Duplicate keys are not allowed
- Duplicate values are just fine

Keys can be any immutable value
numbers, strings, tuples,
not list, dictionary, ...

```
atomic_number_to_name = {
1: "hydrogen"
6: "carbon",
7: "nitrogen"
8: "oxygen",
}
nobel_prize_winners = {
(1979, "physics"): ["Glashow", "Salam", "Weinberg"],
(1962, "chemistry"): ["Hodgkin"],
(1984, "biology"): ["McClintock"],
}
```

More about Dictionary

Get the value for a given key

```
>>> symbol_to_name["C"]
'carbon'>>> "O" in symbol_to_name, "U" in symbol_to_name
(True, False)
```

Test if the key exists
(“in” only checks the keys,
not the values.)

```
>>> "oxygen" in symbol_to_name
False
>>> symbol_to_name["P"]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'P'
>>> symbol_to_name.get("P", "unknown")
'unknown'
>>> symbol_to_name.get("C", "unknown")
'carbon'
```

[] lookup failures raise an exception.
Use “.get()” if you want
to return a default value.

Some useful dictionary methods

```
>>> symbol_to_name.keys()
['C', 'H', 'O', 'N', 'Li', 'He']

>>> symbol_to_name.values()
['carbon', 'hydrogen', 'oxygen', 'nitrogen', 'lithium', 'helium']

>>> symbol_to_name.update( {"P": "phosphorous", "S": "sulfur"} )
>>> symbol_to_name.items()
[('C', 'carbon'), ('H', 'hydrogen'), ('O', 'oxygen'), ('N', 'nitrogen'), ('P', 'phosphorous'), ('S',
    'sulfur'), ('Li', 'lithium'), ('He', 'helium')]

>>> del symbol_to_name['C']
>>> symbol_to_name
{'H': 'hydrogen', 'O': 'oxygen', 'N': 'nitrogen', 'Li': 'lithium', 'He': 'helium'}
```

Control Flow

Things that are False

- The boolean value False
- The numbers 0 (integer), 0.0 (float) and 0j (complex).
- The empty string "".
- The empty list [], empty dictionary {} and empty set set().

Things that are True

- The boolean value True
- All non-zero numbers.
- Any string containing at least one character.
- A non-empty data structure.

Logic

- 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 expressions can be combined with *logical operators*:

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

If statements

```
if (1+1==2):
    print "1+1==2"
    print "I always thought so!"
else:
    print "My understanding of math must be faulty!"
```

Simple one-line if:

```
if (1+1==2): print “I can add!”
```

elif statement

- Equivalent of “else if” in C

```
x = 3
if (x == 1):
    print "one"
elif (x == 2):
    print "two"
else:
    print "many"
```

Use of “If” and “elif” to chain subsequent tests

```
mode = "absolute"
if mode == "canonical":
    smiles = "canonical"
elif mode == "isomeric":
    smiles = "isomeric”
elif mode == "absolute":
    smiles = "absolute"
else:
     print("unknown mode")
```

Boolean logic

Python expressions can have “and”s and “or”s:

```
if (ben <= 5 and chen >= 10 or
chen == 500 and ben != 5):
    print “Ben and Chen“
```

Iteration

- while loops
- for loops
- range function
- Flow control within loops: break, continue, pass.

while

```
i=1
while (i < 4):
    print i
    i += 1
```

Output:

```
1
2
3
```

Function range with for

range(n) returns a list of integers from 0 to n-1.
range(0,10,2) returns a list 0, 2, 4, 6, 8

```
for i in range(3):
    print i,
```

output:

```
 0, 1, 2
```

Flow control within loops

- General structure of a loop:

```
while <statement> (or for <item> in <object>):
    <statements within loop>
    if <test1>: break       # exit loop now
    if <test2>: continue  # go to top of loop now
    if <test3>: pass        # does nothing!
else:
    <other statements> # if exited loop without
                       # hitting a break
```

for … in

- Used with collection data types which can be iterated through (“iterables”):

  ```
  for name in [“Mutasem”, “Micah”, “Ryan”]:
      if name[0] == “M”:
          print (name)
      else:
          print (“Name doesn't start with M”)
  ```
- More about lists and strings later on

Parallel traversals

- If we want to go through 2 lists (more later) in parallel, can use zip:

  ```
  A = [1, 2, 3]
  B = [4, 5, 6]
  for (a,b) in zip(A,B):
    print (a, “*”, b, “=”, a*b)
  ```

  output:

  ```
  1 * 4 = 4
  2 * 5 = 10
  3 * 6  = 18
  ```

Functions

- Define them in the file above the point they're used
- Body of the function should be indented consistently (4 spaces is typical in Python)
- Example:

  ```
  def square(n):
      return n*n

  print ("The square of 3 is “),
  print (square(3))
  ```

  Output:

  ```
  The square of 3 is  9
  ```

The def statement

- The def statement is *excecuted* (that's why functions have to be defined before they're used)
- def creates an object and assigns a name to reference it; the function could be assigned another name, function names can be stored in a list, etc.
- Can put a def statement inside an if statement, etc!

More about functions

- Arguments are optional. Multiple arguments are separated by commas.
- If there's no return statement, then “None” is returned. Return values can be simple types or tuples.
- Functions are “typeless.” Can call with arguments of any type, so long as the operations in the function can be applied to the arguments. This is considered a good thing in Python.

Function variables are local

- Variables declared in a function do not exist outside that function
- Example :

```
def square(n):
    m = n*n
    return (m)

print ("The square of 3 is “),
print  (square(3)),
print  (m)

Output:
  File "./square2.py", line 9, in <module>
    print m
NameError: name 'm' is not defined
```

Scope

- Variables assigned within a function are local to that function call
- Variables assigned at the top of a module are global to that module.

```
a = 5           # global

def func(b):
  c = a + b
  return (c)

print (func(4))       # gives 4+5=9
print (c)             # not defined
```

By value / by reference

- Everything in Python is a reference. However, note also that immutable objects are not changeable --- so changes to immutable objects within a function only change what object the name points to (and do not affect the caller, unless it's a global variable)
- For immutable objects (e.g., integers, strings, tuples), Python acts like C's pass by value
- For mutable objects (e.g., lists), Python acts like C's pass by pointer; in-place changes to mutable objects can affect the caller

Example

```
def f1(x,y):
    x = x * 1
    y = y * 2
    print (x, y)                    # 0 [1, 2, 1, 2]

def f2(x,y):
    x = x * 1
    y[0] = y[0] * 2
    print (x, y)                    # 0 [2, 2]

a = 0
b = [1,2]
f1(a,b)
print (a, b)                        # 0 [1, 2]
f2(a,b)
print (a, b)                        # 0 [2, 2]
```

Multiple return values

- Can return multiple values by packaging them into a tuple

```
def onetwothree(x):
  return (x*1, x*2, x*3)

print (onetwothree(3))
```

```
3, 6, 9
```

Default arguments

- Like C or Java, can define a function to supply a default value for an argument if one isn't specified

```
def print_error(lineno, message=”error”):
  print (message)

print_error(42)
```

- Output:
error

Functions without return values

- All functions in Python return something. If a return statement is not given, then by default, Python returns None
- Beware of assigning a variable to the result of a function which returns None. For example, the list append function changes the list but does not return a value:

  ```
  a = [0, 1, 2]
  b = a.append(3)
  print b
  None
  ```

Classes

- Classes are defined using the **`class`** statement

-
  ```
  >>> class Foo:
  ...   def __init__(self):
  ...     self.member = 1
  ...   def GetMember(self):
  ...     return self.member
  >>>
  ```

  The constructor has a special name **__init__**

  The **self** parameter is the instance (ie, the **this** in C++).

Encapsulation

“Encapsulation can be used to hide data members and members function. Under this definition, encapsulation means that the internal representation of an object is generally hidden from view outside of the object’s definition.” —Wikipedia

```
class Person:
  def __init__(self, first_name):
        self.first_name = first_name
```

Encapsulation

- *"'Private' instance variables that cannot be accessed except from inside an object don't exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g.* _spam*) should be treated as a non-public part of the API (whether it is a function, a method or a data member)"—[Python Software Foundation]()*

Encapsulation

```
class Person:
    def __init__(self, first_name, email):
        self.first_name = first_name
        self._email = email
    def update_email(self, new_email):
        self._email = new_email
    def email(self):
        return self._email
```

But there is a method in Python to define Private: Add “__” (double underscore) in front of the variable and function name can hide them when accessing them from out of class.

```
Class SeeMee:
  def youcanseeme(self):
    return 'you can see me'

  def __youcannotseeme(self):
    return 'you cannot see me'

#Outside class
Check = SeeMee()
print(Check.youcanseeme())

print(Check.__youcannotseeme())
#AttributeError: 'SeeMee' object has no attribute
'__youcannotseeme‘
```

But still you can figure out the way to access private member

```
print(Check._SeeMee__youcannotseeme())
```

Inheritance

```
class Person:
    def __init__(self, first, last):
            self.firstname = first
            self.lastname = last
    def Name(self):
            return (self.firstname + " " + self.lastname)
class Employee(Person):
     def __init__(self, first, last, staffnum):
            Person.__init__(self,first, last)
            self.staffnumber = staffnum
     def GetEmployee(self):
            return (self.staffnumber, self.Name())
```

Overriding in Python

```
class Parent(object):
       def __init__(self):
             self.value = 5
      def get_value(self):
              return (self.value)
class Child(Parent):
       def get_value(self):
              return (self.value + 1)
>>> c = Child()
>>> c.get_value()
6
```

Overloading

```
class  Example:
    def __init__(self):
       pass
    def product(self,a, b):
       p = a * b
       print(p)
   def product(self,a, b, c):
     p = a * b*c
     print(p)
>>> c = example()
>>> c. product(2,3)
6
>>> c. product(2,3,4)
24
```

Operator Overloading

- **operator overloading**: You can define functions so that Python's built-in operators can be used with your class.

Operator	Class Method
-	__neg__(self, other)
+	__pos__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)

-	__neg__(self)
+	__pos__(self)

Operator	Class Method
==	__eq__(self, other)
!=	__ne__(self, other)
<	__lt__(self, other)
>	__gt__(self, other)
<=	__le__(self, other)
>=	__ge__(self, other)

Polymorphism

```
class Shark():
    def skeleton(self):
        print("The shark's skeleton is made of cartilage.")
class Clownfish():
    def skeleton(self):
        print("The clownfish's skeleton is made of bone.")

sammy = Shark()
casey = Clownfish()
for fish in (sammy, casey):
    fish.skeleton()
```