Soongsil University

2150686601
이산수학

Introduction to Python Programming

목차

Why Use Python? (1)

- Python is object-oriented
  - Structure supports such concepts as polymorphism, operation overloading, and multiple inheritance
- It's free (open source)
  - Downloading and installing Python is free and easy
  - Source code is easily accessible
  - Free doesn't mean unsupported! Online Python community is huge
- It's portable
  - Python runs virtually every major platform used today
  - As long as you have a compatible Python interpreter installed, Python programs will run in exactly the same manner, irrespective of platform
- It's powerful
  - Dynamic typing
  - Built-in types and tools
  - Library utilities
  - Third party utilities (e.g. NumPy, SciPy)
  - Automatic memory management

Why Use Python? (2)

- It's mixable
  - Python can be linked to components written in other languages easily
    - ✓ Linking to fast, compiled code is useful to computationally intensive problems
    - ✓ Python is good for code steering and for merging multiple programs in otherwise conflicting languages
  - Python/C integration is quite common
  - WARP is implemented in a mixture of Python and Fortran
- It's easy to use
  - Rapid turnaround: no intermediate compile and link steps as in C or C++
  - Python programs are compiled automatically to an intermediate form called *bytecode*, which the interpreter then reads
  - This gives Python the development speed of an interpreter without the performance loss inherent in purely interpreted languages
- It's easy to learn
  - Structure and syntax are pretty intuitive and easy to grasp

Running Python (1)

- How to call up a Python interpreter will vary a bit depending on your platform, but in a system with a terminal interface, all you need to do is type “python” (without the quotation marks) into your command line.

Example:

# From here on, the $ sign denotes the start of a terminal command line, and the # sign denotes a comment. Note: the # sign denotes a comment in Python. Python ignores anything written to the right of a # sign on a given line

```
$ python     # Type python into your terminal's command line
>>>
             # After a short message, the >>> symbol will appear.  This signals
             # the start of a Python interpreter's command line
```

Running Python (2)

Once you're inside the Python interpreter, type in commands at will.

Examples:

```
>>> print ('Hello world')
Hello world
# Relevant output is displayed on subsequent lines without the >>> symbol
>>> x = [0,1,2]
# Quantities stored in memory are not displayed by default

# If a quantity is stored in memory, typing its name will display it
>>> x
[0,1,2]
>>> 2+3
5
>>>  # Type ctrl-Z to exit the interpreter (at Windows Command Shell)
$ (or C:\Users\CST)
```

Running Python (3)

Python scripts can be written in text files with the suffix .py. The scripts can be read into the interpreter in several ways:

Examples:

```
$ python script.py
```

# This will simply execute the script and return to the terminal afterwards

```
$ python -i script.py
```

# The -i flag keeps the interpreter open after the script is finished running

```
$ python -i Hello.py
…………….
>>> exec(open(“./hellow.py”).read())
```

# The exec command execute the opened script immediately, as though they had been typed into the interpreter directly

```
$ python
>>> import script # DO NOT add the .py suffix.  Script is a module here
```

# The import command runs the script, displays any unstored outputs, and creates a lower level (or context) within the program. More on contexts later.

Running Python (4)

Suppose the file script.py contains the following lines:

```
print (“Hello world”)
x = [0,1,2]
```

Let's run this script in each of the ways described on the last slide:

Examples:

```
$ python script.py
Hello world
$
```

# The script is executed and the interpreter is immediately closed. x is lost.

```
$ python -i script.py
Hello world
>>> x
[0,1,2]
>>>
```

# “Hello world” is printed, x is stored and can be called later, and the interpreter is left open

Running Python (5)

Examples: (continued from previous slide)

```
$ python
>>> exec(open('script.py‘).read())
Hello world
>>> x
[0,1,2]
>>>
```

# For our current purposes, this is identical to calling the script from the terminal with the command `python -i script.py`

```
$ python
>>> import script
Hello world
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'x' is not defined
>>>
```

# When script.py is loaded in this way, x is not defined on the top level

Running Python (6)

Examples: (continued from previous slide)

# to make use of x, we need to let Python know which module it came from, i.e. give Python its context

```
>>> script.x
[0,1,2]
>>>
```

# Pretend that script.py contains multiple stored quantities. To promote x (and only x) to the top level context, type the following:

```
$ python
>>> from script import x
Hello world
>>> x
[0,1,2]
>>>
```

# To promote all quantities in script.py to the top level context, type `from script import *` into the interpreter. Of course, if that's what you want, you might as well type `python -i script.py` into the terminal.

Numbers (1)

Python supports several different numeric types

- Integers
  - Examples: `0, 1, 1234, -56`
  - Integers are implemented as C longs
  - Note: dividing an integer by another integer will return only the
  - integer part of the quotient, e.g. typing `7/2` will yield `3`
- Floating point numbers
  - Examples: `0., 1.0, 1e10, 3.14e-2, 6.99E4, float(3)`
  - Implemented as C doubles
  - Division works normally for floating point numbers: `7./2. = 3.5`
  - Operations involving both floats and integers will yield floats: `6.4 - 2 = 4.4`

Numbers (2)

Other numeric types:

- Binary constants
  - Examples: `0b111, -0b0101`
  - Must start with a leading `0b`
- Octal constants
  - Examples: `0o177, -0o1234`
  - Must start with a leading `0o`
- Hex constants
  - Examples: `0x9ff, 0X7AE`
  - Must start with a leading `0x` or `0X`
- Complex numbers
  - Examples: `3+4j, 3.0+4.0j, 2J`
  - Must end in `j` or `J`
  - Typing in the imaginary part first will return the complex number in the order `Re+ImJ`
  - `Complex(3,5)`

Numbers (3)

https://www.w3schools.com/python/python_operators.asp

- Basic algebraic operations
  - Four arithmetic operations: `a+b, a-b, a*b, a/b`
  - Exponentiation: `a**b (=` a^b)
  - Other elementary functions are not part of standard Python, but included in packages like NumPy and SciPy
- Comparison operators
  - Greater than, less than, etc.: `a < b, a > b, a <= b, a >= b`
  - Identity tests: `a == b, a != b`
- Bitwise operators
  - Bitwise or: `a | b`
  - Bitwise exclusive or: `a ^ b` # Don't confuse this with exponentiation
  - Bitwise and: `a & b`
  - Shift `a` left or right by `b` bits: `a << b, a >> b`
- Other
  - Not surprisingly, Python follows the basic PEMDAS order of operations (**P**arentheses first, **E**xponents(Powers and Square Roots, etc), **M**ultiply, |**D**ivide, **A**dd|**S**ubtract)
  - Python supports mixed-type math. The final answer will be of the most complicated type used.

Text Sequence Type : Strings

Strings are ordered blocks of text

- Strings are enclosed in single or double quotation marks
- Double quotation marks allow the user to extend strings over multiple multiple lines without backslashes, which usually signal the continuation continuation of an expression
- Python has no “character” type ; it treats a single character as the same same as a one-character string.
- Examples: 'abc', “ABC”
- There is a function to convert from other types, such as integers and and floating point numbers, to strings: its name is str. For example, str(3)

Indexing and Slicing of Strings (1)

Indexing and slicing

- Python starts indexing at `0`. A string `s` will have indexes running from `0` to `len(s)-1` (where `len(s)` is the length of `s`) in integer quantities.
- `s[i]` fetches the ith element in s

```
>>> s = 'string'
>>> s[1]  # note that Python considers 't' the first element
't'       # of our string s
```

- `s[i:j]` fetches elements i (inclusive) through j (not inclusive)

```
>>> s[1:4]
'tri'
```

- `s[:j]` fetches all elements up to, but not including j

```
>>> s[:3]
'str'
```

- `s[i:]` fetches all elements from i onward (inclusive)

```
>>> s[2:]
'ring'
```

Indexing and Slicing of Strings (2)

Indexing and slicing, contd.

- `s[i:j:k]` extracts every `k`th element starting with index `i` (inlcusive) and ending with index `j` (not inclusive)

```
>>> s[0:5:2]
'srn'
```

- Python also supports negative indexes. For example, `s[-1]` means extract the first element of `s` from the end (same as `s[len(s)-1]`)

```
>>> s[-1]
'g'
>>> s[-2]
'n'
```

- Python's indexing system is different from those of Fortran, MatLab, and Mathematica. The latter three programs start indexing at 1, and have inclusive slicing, i.e. the last index in a slice command is included in the slice

Sequence type : Lists

Basic properties:

- Lists are contained in square brackets `[]`
- Lists can contain numbers, strings, nested sublists, or nothing
- Examples: `L1 = [0,1,2,3]`, `L2 = ['zero', 'one']`, `L3 = [0,1,[2,3],'three',['four,one']]`, `L4 = []`
- List indexing works just like string indexing
- Lists are mutable: individual elements can be reassigned in place.
- Moreover, they can grow and shrink in place
- Example:

```
>>> L1 = [0,1,2,3]
>>> L1[0] = 4
>>> L1[0]
4
```

Operations on Lists (1)

Some basic operations on lists:

- Indexing: `L1[i], L2[i][j]`
- Slicing: `L3[i:j]`
- Concatenation:

```
>>> L1 = [0,1,2]; L2 = [3,4,5]
>>> L1+L2
[0,1,2,3,4,5]
```

- Repetition:

```
>>> L1*3
[0,1,2,0,1,2,0,1,2]
```

- Appending:

```
>>> L1.append(3)
[0,1,2,3]
```

- Sorting:

```
>>> L3 = [2,1,4,3]
>>> L3.sort()
>>> L3
[1,2,3,4]
```

Operations on Lists (2)

More list operations:

- Reversal:

```
>>> L4 = [4,3,2,1]
>>> L4.reverse()
>>> L4
[1,2,3,4]
```

- Shrinking:

```
>>> del L4[2]
>>> L4[i:j] = []
```

- Index and slice assignment:

```
>>> L1[1] = 1
>>> L2[1:4] = [4,5,6]
```

- Making a list of integers:

```
>>> list(range(4))
[0,1,2,3]
>>> list(range(1,5))
[1,2,3,4]
```

Sequence Type : Tuples

Basic properties:

- A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. The important difference is that tuples are immutable.
- Tuples are usually contained in parentheses `()`
- Tuples can contain numbers, strings, nested sub-tuples, or nothing
- Examples: `t1 = (0,1,2,3), t2 = ('zero', 'one'), t3 = (0,1,(2,3),'three',('four,one')), t4 = ()`
- As long as you're not nesting tuples, you can omit the parentheses
  Example: `t1 = 0,1,2,3` is the same as `t1 = (0,1,2,3)`
- Tuple indexing works just like string and list indexing
- Concatenation:

```
>>> t1 = (0,1,2,3); t2 = (4,5,6)
>>> t1+t2
(0,1,2,3,4,5,6)
```

- Repetition:

```
>>> t1*2
(0,1,2,3,0,1,2,3)
```

- Length: `len(t1)` (this also works for lists and strings)

Sequence Type : range

- *class* range(*stop)*
  - *list(range (4)) → [0,1,2,3]*

- *class* range(*start*, *stop*[, *step*])
  - *list(range(1,5)) → [1,2,3,4]*
  - list(range(1,6,2)) → [1, 3, 5]

Sets

- 집합 자료형 ; no repetition. Unordered.
- 집합 생성
  - setA={1,2,3}
  - setB =set([3,4,5])
  - setC = set("Hello")

    >>> setC ={'e', 'H', 'l', 'o'}
  - { *n* | 0 < *n* < 100 and *n* is odd }

  → { n for n in range (100) if n%2 ==1 }
- Iteration over set

```
cities = { "London", "Paris", "Vienna", "Istanbul" }
for place in cities:
    print(place)
```

Sets Operation

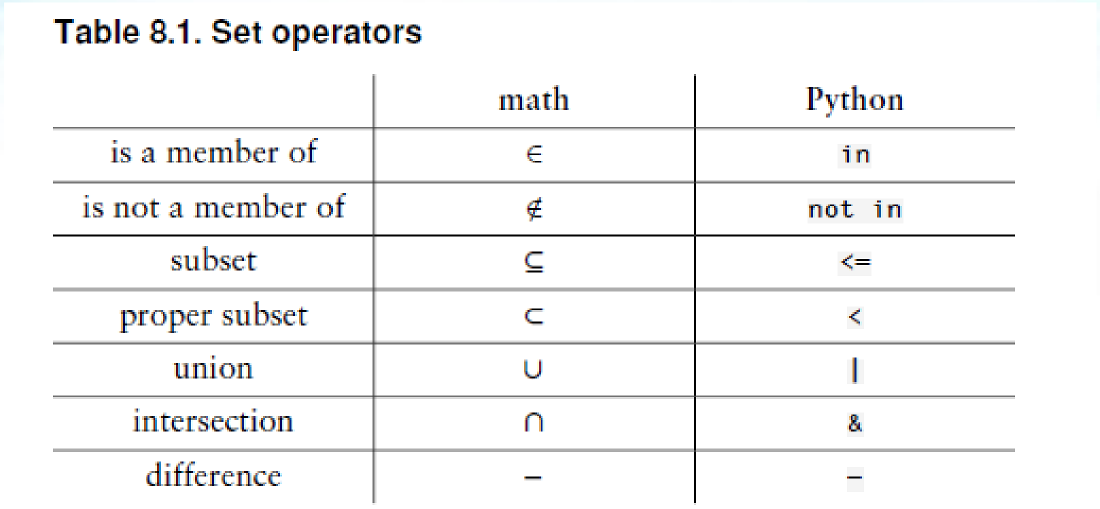


Table 8.1. Set operators

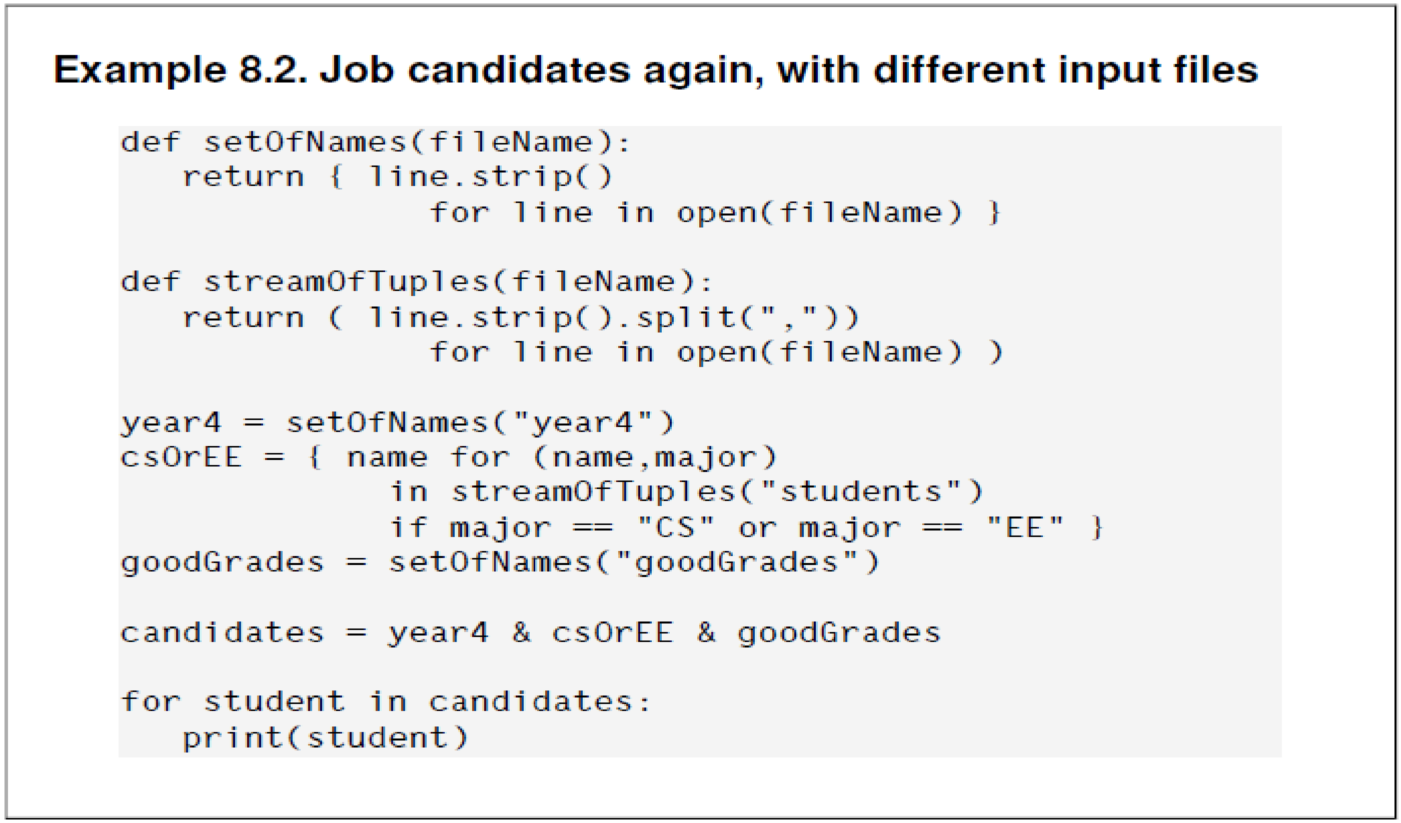
	math	Python
is a member of	$\in$	`in`
is not a member of	$\notin$	`not in`
subset	$\subseteq$	`<=`
proper subset	$\subset$	`<`
union	$\cup$	`|`
intersection	$\cap$	`&`
difference	$-$	`-`

- 집합연산 ; setA ={ 1,2,3}, setB= {3,4,5}
  - 합집합 ; setA | setB → {1,2,3,4,5}
  - 교집합 ; setA & setB → {3}
  - 차집합 ; setA – setB → { 1, 2}
  - is a member

```
>>>3 in setA
    True
```

A case study: finding students for jobs

- Suppose that a company is recruiting students at a university to give them jobs after they graduate, and suppose that the company wants to interview fourth-year students who are either computer science or electrical engineering students and who have a grade average of B or better
- 3 files ; “students” (Jim,CS …) , “year4”, “goodGrades”



Example 8.2. Job candidates again, with different input files

```
def setOfNames(fileName):
   return { line.strip()
               for line in open(fileName) }

def streamOfTuples(fileName):
   return ( line.strip().split(","))
               for line in open(fileName) )

year4 = setOfNames("year4")
csOrEE = { name for (name,major)
             in streamOfTuples("students")
             if major == "CS" or major == "EE" }
goodGrades = setOfNames("goodGrades")

candidates = year4 & csOrEE & goodGrades

for student in candidates:
   print(student)
```

Dictionary type

- {Key1:Value1, Key2:Value2, Key3:Value3 ...}
- 리스트나 튜플처럼 순차적으로(sequential) 해당 요소값을 구하지 않고 Key를 통해 Value를 얻는다.
  - ex. dic = {'alpha' : 3, 'beta' : 4 , 'gamma' : 'foo' }

```
dic['alpha']          dic['gamma']
  3                   'foo'
  >>>                   >>>
```

Arrays from Numpy (1)

Note: Arrays supported in the built-in python standard library is not powerful ; Arrays included in the third-party package, NumPy, is very powerful.

- NumPy is a general-purpose array-processing package designed to efficiently manipulate large multi-dimensional arrays of arbitrary records without sacrificing too much speed for small multi-dimensional arrays.

Basic useage:

- Loading in array capabilities: # from here on, all operations involving arrays assume you have already made this step
```
>>> from numpy import *
```
- Creating an array:
```
>>> vec = array([1,2,3])
```
- Creating a 3x3 matrix:
```
>>> mat = array([[1,2,3],[4,5,6],[7,8,9]])
```
- If you need to initialize a dummy array whose terms will be altered later, the `zeros` and `ones` commands are useful; `zeros((m,n))` will create an m-by-n array of zeros, which can be integers, floats, double precision floats etc. depending on the type code used

Arrays from Numpy (2)

Arrays and lists have many similarities, but there are also some important differences

Similarities between arrays and lists:

- Both are mutable: both can have elements reassigned in place
- Arrays and lists are indexed and sliced identically
- The `len` command works just as well on arrays as anything else
- Arrays and lists both have `sort` and `reverse` attributes
- Differences between arrays and lists:
- With arrays, the + and * signs do not refer to concatenation or repetition
- Examples:

```
>>> ar1 = array([2,4,6])
>>> ar1+2  # Adding a constant to an array adds the constant to each term
[4,6,8]    # in the array
>>> ar1*2  # Multiplying an array by a constant multiplies each term in
[4,8,12]  # the array by that constant
```

Arrays from Numpy (3)

More differences between arrays and lists:

- Adding two arrays is just like adding two vectors

```
>>> ar1 = array([2,4,6]); ar2 = array([1,2,3])
>>> ar1+ar2
[3,6,9]
```

- Multiplying two arrays multiplies them term by term:

```
>>> ar1*ar2
[2,8,18]
```

- Same for division:

```
>>> ar1/ar2
[2,2,2]
```

- Assuming the function can take vector arguments, a function acting on an array acts on each term in the array

```
>>> ar2**2
[1,4,9]
>>> ar3 = (pi/4)*arange(3)  # like range, but an array
>>> sin(ar3)
[ 0.        , 0.70710678, 1. ]
```

Arrays from Numpy (4)

More differences between arrays and lists:

- The biggest difference between arrays and lists is speed; it's much faster to carry out operations on arrays (and all the terms therein) than on each term in a given list.
  Example: take the following script:

```
from math import *
From numpy import *
import time
tt1 = time.clock()
sarr = 1.*arange(0,10001)/10000
sinarr = sin(sarr)
tt2 = time.clock()
slist = []; sinlist = []
for i in range(10001):
  slist.append(1.*i/10000)
  sinlist.append(sin(slist[i]))
tt3 = time.clock()
```

- Running this script on my system shows that `tt2-tt1` (i.e., the time it takes to set up the array and take the sin of each term therein) is 0.0 seconds, while `tt3-tt2` (the time to set up the list and take the sin of each term therein) is 0.26 seconds.

Mutable vs. Immutable Types

Mutable types (dictionaries, lists, arrays) can have individual items reassigned in place, while immutable types (strings, tuples) cannot.

```
>>> L = [0,2,3]
>>> L[0] = 1
>>> L
[1,2,3]
>>> s = 'string'
>>> s[3] = 'o'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
```

However, there is another important difference between mutable and immutable types; they handle name assignments differently. If you assign a name to an immutable item, then set a second name equal to the first, changing the value of the first name will not change that of the second. However, for mutable items, changing the value of the first name will change that of the second.
An example to illustrate this difference follows on the next slide.

Basic Statements: The If Statement (1)

If statements have the following basic structure:

```
# inside the interpreter                  # inside a script
>>> if condition:                     if condition:
...     action                          action
...
>>>
```

Subsequent indented lines are assumed to be part of the if statement. The same is true for most other types of python statements. A statement typed into an interpreter ends once an empty line is entered, and a statement in a script ends once an unindented line appears. The same is true for defining functions.

If statements can be combined with else if (elif) and else statements as follows:

```
if condition1:       # if condition1 is true, execute action1
  action1
elif condition2:     # if condition1 is not true, but condition2 is, execute
  action2            # action2
else:                # if neither condition1 nor condition2 is true,
execute
  action3            # action3
```

Basic Statements: The If Statement (2)

Conditions in if statements may be combined using `and` & `or` statements

```
if condition1 and condition2:
  action1
```

# if both condition1 and condition2 are true, execute action1

```
if condition1 or condition2:
  action2
```

# if either condition1 or condition2 is true, execute action2

Conditions may be expressed using the following operations:

```
<, <=, >, >=, ==, !=, in
```

Somewhat unrealistic example:

```
>>> x = 2; y = 3; L = [0,1,2]
>>> if (1<x<=3 and 4>y>=2) or (1==1 or 0!=1) or 1 in L:
...   print ('Hello world')
...
Hello world
>>>
```

Basic Statements: The While Statement (1)

While statements have the following basic structure:

```
# inside the interpreter                    # inside a script
>>> while condition:                       while condition:
...     action                                 action
...
>>>
```

As long as the condition is true, the while statement will execute the action

Example:

```
>>> x = 1
>>> while x < 4:    # as long as x < 4...
...     print x**2  # print the square of x
...     x = x+1     # increment x by +1
...
1                   # only the squares of 1, 2, and 3 are printed, because
4                   # once x = 4, the condition is false
9
>>>
```

Basic Statements: The While Statement (2)

Pitfall to avoid:

While statements are intended to be used with changing conditions. If the condition in a while statement does not change, the program will be stuck in an infinite loop until the user hits ctrl-C.

Example:

```
>>> x = 1
>>> while x == 1:
...    Print('Hello world‘)
...
```

Since x does not change, Python will continue to print “Hello world” until interrupted

Basic Statements: The For Statement (1)

For statements have the following basic structure:

```
for item i in set s:
   action on item i
```

# item and set are not statements here; they are merely intended to clarify the relationships between i and s

Example:

```
>>> for i in range(1,7):
...    print (i, i**2, i**3, i**4)
...
1 1 1 1
2 4 8 16
3 9 27 81
4 16 64 256
5 25 125 625
6 36 216 1296
>>>
```

Basic Statements: The For Statement (2)

The item i is often used to refer to an index in a list, tuple, or array
Example:

```
>>> L = [0,1,2,3]   # or, equivalently, range(4)
>>> for i in range(len(L)):
...   L[i] = L[i]**2
...
>>> L
[0,1,4,9]
>>>
```

Of course, we could accomplish this particular task more compactly using arrays:

```
>>> L = arange(4)
>>> L = L**2
>>> L
[0,1,4,9,]
```

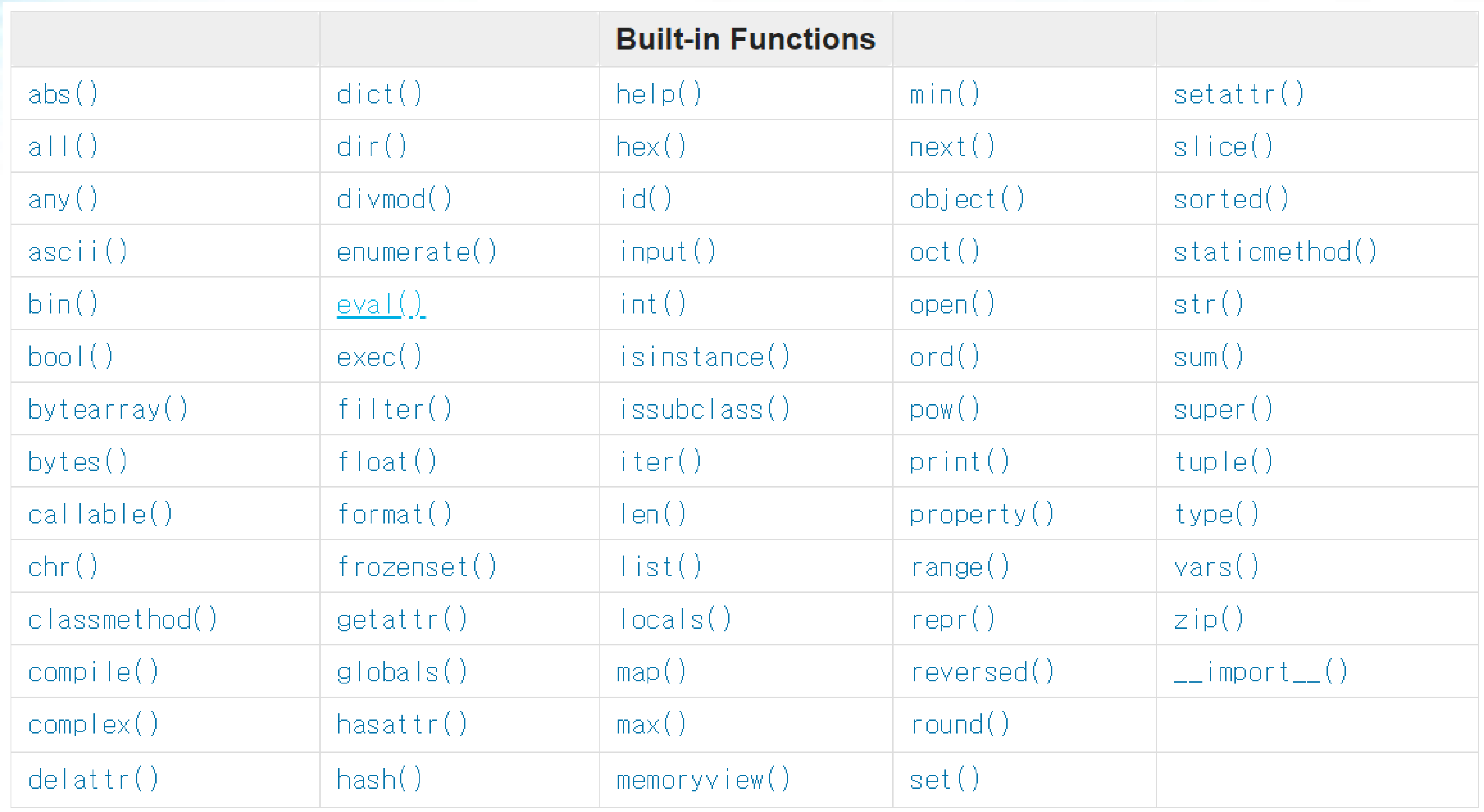
Basic Statements: Combining Statements

The user may combine statements in a myriad of ways

Example:

```
>>> L = [0,1,2,3]   # or, equivalently, range(4)
>>> for i in range(len(L)):
...   j = i/2.
...   if j - int(j) == 0.0:
...     L[i] = L[i]+1
...   else: L[i] = -i**2
...
>>> L
[1,-1,3,-9]
>>>
```

Built-in Functions



		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Functions (1)

Usually, function definitions have the following basic structure:

```
def func(args):
   return values
```

Regardless of the arguments, (including the case of no arguments) a function call must end with parentheses.

Functions may be simple one-to-one mappings

```
>>> def f1(x):
...    return x*(x-1)
...
>>> f1(3)
6
```

They may contain multiple input and/or output variables

```
>>> def f2(x,y):
...    return x+y,x-y
...
>>> f2(3,2)
(5,1)
```

Functions (2)

Functions don't need to contain arguments at all:

```
>>> def f3():
...    print 'Hello world'
...
>>> f3()
Hello world
```

The user can set arguments to default values in function definitions:

```
>>> def f4(x,a=1):
...    return a*x**2
...
>>>
```

If this function is called with only one argument, the default value of 1 is assumed for the second argument

```
>>> f4(2)
4
```

However, the user is free to change the second argument from its default value

```
>>> f4(2,a=2)   # f4(2,2) would also work
8
```

Functions (3)

Functions need not take just numbers as arguments, nor output just numbers or tuples. Rather, they can take multiple types as inputs and/or outputs.

Examples:

```
>>> arr = arange(4)
>>> f4(arr,a=2)  # using the same f4 as on the previous slide
[0,2,8,18,]
>>> def f5(func, list, x):
...    L = []
...    for i in range(len(list)):
...      L.append(func(x+list[i]))
...    arr = array(L)
...    return L,arr
...
>>> L1 = [0.0,0.1,0.2,0.3]
>>> L,arr = f5(exp,L1,0.5)
>>> arr
[ 1.64872127, 1.8221188 , 2.01375271, 2.22554093,]
```

Note: the function above requires Numeric, NumPy, or a similar package

Functions (4)

Anything calculated inside a function but not specified as an output quantity (either with `return` or `global`) will be deleted once the function stops running

```
>>> def f5(x,y):
...    a = x+y
...    b = x-y
...    return a**2,b**2
...
>>> f5(3,2)
(25,1)
```

If we try to call a or b, we get an error message:

```
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'a' is not defined
```

This brings us to scoping issues, which will be addressed in the next section.

Functions (5)

Lambda expression ; have functions as their values
python supports lambda expression

```
>>> def square(x):
...    return x*x
...
>>> square = lambda x: x*x
>>> square(3)
9
>>>
```

In Python a function can even return another function as a value. A function can can execute a function-definition statement, creating a new function and binding binding that function to a name, and then return the value bound to that name.

```
>>>def partial(f, x):
...     def f_x (y):
...             return f(x,y)
...     return f_x
>>> double=partial(lambda x,y:x*y , 2)
>>> double(3)
```

Functions: Getting Help

If you forget how to use a standard function, Python's library utilities can help. Say we want to know how to use the function `execfile()`. In this case, Python's `help()` library functions is extremely relevant.

Usage:

```
>>> help(exec)
```

# don't include the parentheses when using the function name as an argument

Entering the above into the interpreter will call up an explanation of the function, its usage, and the meanings of its arguments and outputs. The interpreter will disappear and the documentation will take up the entire terminal. If the documentation takes up more space than the terminal offers, you can scroll through the documentation with the up and down arrow keys. Striking the q key will quit the documentation and return to the interpreter.

Higher-order Sequence functions(1)

- Higher-order functions ; functions that act on or return other functions
- Python supports a classic trio of higher-order functions – map, filter, reduce

map

- applies a function to all the items in an input_list.

```
>>> items = [1, 2, 3, 4, 5]
>>>  squared = list(map(lambda x: x**2, items))
>>> squared
[1, 4, 9, 16, 25]
```

filter

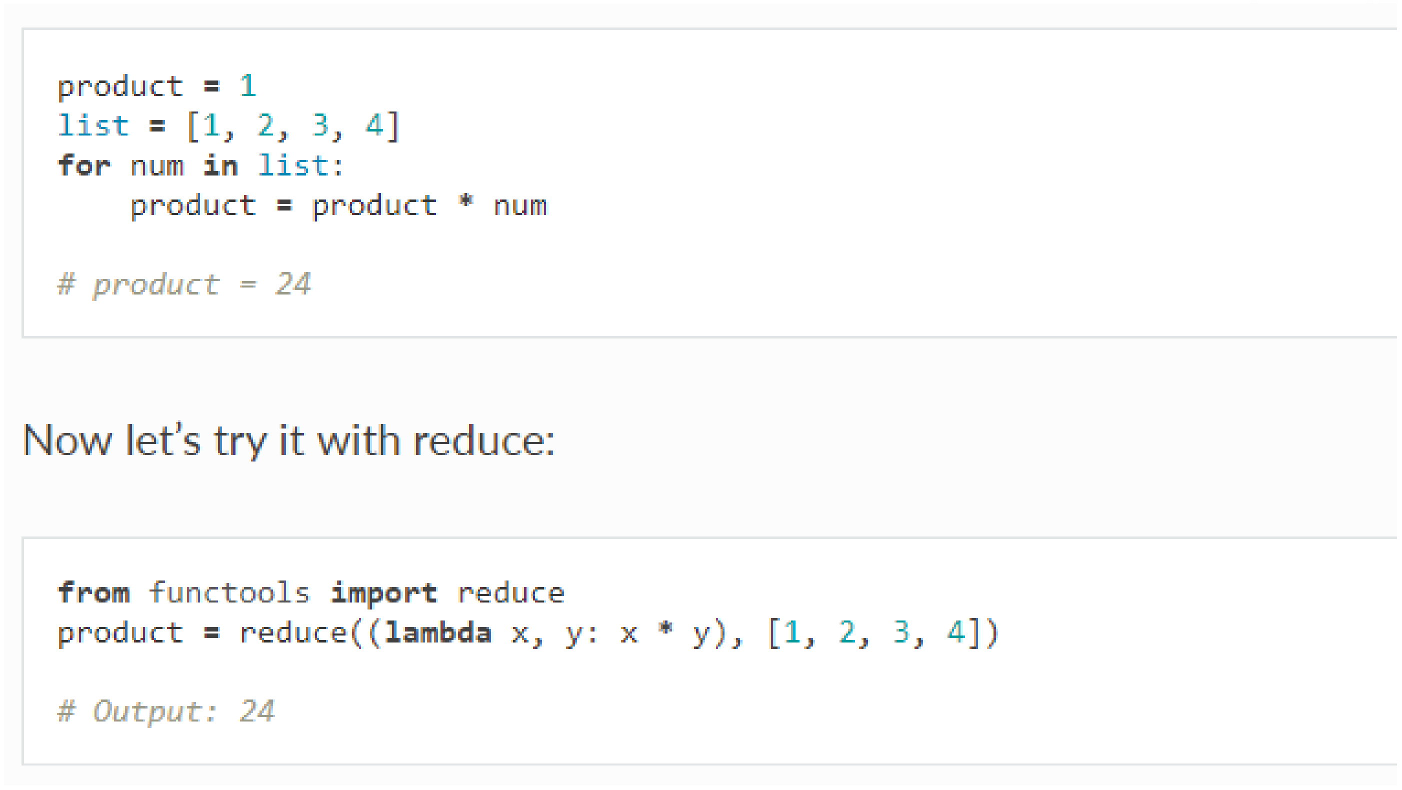
- creates a list of elements for which a function returns true

```
>>> number_list = range(-5, 5)
>>> less_than_zero = list(filter(lambda x: x < 0, number_list))
>>> less_than_zero
  [-5, -4, -3, -2, -1]
```

Higher-order Sequence functions(2)

- ‘reduce’ is a really useful function for performing some computation on a list and returning the result. It applies a rolling computation to sequential pairs of values in a list.
- ‘reduce’ is not supported as a built-in function. ‘functools’

■ reduce



```
product = 1
list = [1, 2, 3, 4]
for num in list:
    product = product * num

# product = 24
```

Now let's try it with reduce:

```
from functools import reduce
product = reduce((lambda x, y: x * y), [1, 2, 3, 4])

# Output: 24
```

List compression

- A list comprehension constructs a list from one or more other sequences, applying function application and filtering in much the same way as the map and filter functions do, but with a syntax that perhaps suggests the structure of the result more directly.
- Here is the most basic form of a list comprehension:

  [expression for name in sequence]
- Ex.

```
 [ 2**n for n in range(5) ]
list(map(lambda n: 2**n, range(5)))
 [ (a,b) for a in range(6) for b in range(5) ]
```

- A comprehension can also filter the sequence that it draws values from. Then the syntax is

  [*expression* for *name* in *sequence* if *test*]
- Ex.

```
[ line[5:].strip() for line in open("names") if line.startswith("John ") ]
```

Scope Rules (1)

Python employs the following scoping hierarchy, in decreasing order of breadth:

- Built-in (Python)
  - Predefined names (len, open, exec, etc.) and types
- Global (module)
  - Names assigned at the top level of a module, or directly in the interpreter
  - Names declared global in a function
- Local (function)
  - Names assigned inside a function definition or loop

Example:

```
>>> a = 2          # a is assigned in the interpreter, so it's global
>>> def f(x):      # x is in the function's argument list, so it's local
...    y = x+a     # y is only assigned inside the function, so it's local
...    return y    # using the sa
...
>>>
```

Scope Rules (2)

If a module file is read into the interpreter via execfile, any quantities defined in the top level of the module file will be promoted to the top level of the program

As an example: return to our friend from the beginning of the presentation, script.py:

```
print ('Hello world')
x = [0,1,2]
>>> exec(open('script.py').read())
Hello world
>>> x
 [0,1,2]
```

If we had imported script.py instead, the list x would not be defined on the top level. To call x, we would need to explicitly tell Python its scope, or context.

```
>>> import script
Hello world
>>> script.x
 [0,1,2]
```

As we saw on slide 9, if we had tried to call x without a context flag, an error message would have appeared

Scope Rules (3)

Modules may well contain submodules. Say we have a file named module.py which, in its definition, imports a submodule named submodule, which in turn contains some quantity named x.

```
>>> import module
```

If we load the module this way, we would type the following to call x:

```
>>> module.submodule.x
```

We can also import the submodule without importing other quantities defined in module.py:

```
>>> from module import submodule
```

In this case, we would type the following to call x:

```
>>> submodule.x
```

We would also call x this way if we had read in module.py with `exec()`

Scope Rules (4)

You can use the same names in different scopes
Examples:
```
>>> a = 2
>>> def f5(x,y)
...    a = x+y     # this a has no knowledge of the global a, and vice-versa
...    b = x-y
...    return a**2,b**2
...
>>> a
2
```
The local a is deleted as soon as the function stops running
```
>>> x = 5
>>> import script   # same script as before
Hello world
>>> x
5
>>> script.x        # script.x and x are defined in different scopes, and
[0,1,2]             # are thus different
```

Scope Rules (5)

Changing a global name used in a function definition changes the function
Example:

```
>>> a = 2
>>> def f(x):
...     return x+a    # this function is, effectively, f(x) = x+2
...
>>> f(4)
6
>>> a = 1
>>> f(4)              # since we set a=1, f(x) = x+1 now
5
```

Unlike some other languages, Python function arguments are not modified by default:

```
>>> x = 4
>>> f(x)
5
>>> x
4
```

Python Standard library

- Python's standard library is very extensive, offering a wide range of facilities as indicated by the long table of contents listed below.
- The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming.
- Some of these modules are explicitly designed to encourage and enhance the portability of Python programs by abstracting away platform-specifics into platform-neutral APIs.
- In addition to the standard library, there is a growing collection of several thousand components (from individual programs and modules to packages and entire application development frameworks), available from the Python Package Index.

What are Modules?

- Modules are files containing Python definitions and statements (ex. *name*.py)
- A module's definitions can be imported into other modules by using "import *name*"
- The module's name is available as a global variable value
- To access a module's functions, type *"name.function()"*

More on Modules

- Modules can contain executable statements along with function definitions
- Each module has its own private symbol table used as the global symbol table by all functions in the module
- Modules can import other modules
- Each module is imported once per interpreter session
  - reload(name)
- Can import names from a module into the importing module's symbol table
  - *from mod import m1, m2* (or ***)
  - *m1()*

Executing Modules

- *python name.py <arguments>*
  - Runs code as if it was imported
  - Setting *_name_ == "_main_" the file can be used as a script and an importable module*

The Module Search Path

- The interpreter searches for a file named *name.py*
  - Current directory given by variable *sys.path*
  - List of directories specified by PYTHONPATH
  - Default path
    - in UNIX ; *.:/usr/local/lib/python*
    - *In Windows ;*
- Script being run should not have the same name as a standard module or an error will occur when the module is imported

“Compiled” Python Files

- If files *mod.pyc* and *mod.py* are in the same direc tory, there is a byte-compiled version of the mod ule *mod*
- The modification time of the version of *mod.py* us ed to create *mod.pyc* is stored in *mod.pyc*
- Normally, the user does not need to do anything t o create the *.pyc* file
- A compiled *.py* file is written to the *.pyc*
  - No error for failed attempt, *.pyc* is recognized as inv alid
- Contents of the *.pyc* can be shared by different machines

Some Tips

- –*O* flag generates optimized code and stores it in *.pyo* files
  - Only removes *assert* statements
  - *.pyc* files are ignored and *.py* files are compiled to optimized bytecode
- Passing two –*OO* flags
  - Can result in malfunctioning programs
  - _*doc*_ strings are removed
- Same speed when read from *.pyc, .pyo,* or *.py* files, *.pyo* and *.pyc* files are loaded faster
- Startup time of a script can be reduced by moving its code to a module and importing the module
- Can have a *.pyc* or *.pyo* file without having a *.py* file for the same module
- Module *compileall* creates *.pyc* or *.pyo* files for all modules in a directory

Standard Modules

- Python comes with a library of standard modules described in the Python Library Reference
- Some are built into interpreter

```
>>> import sys
>>> sys.s1
'>>> '
>>> sys.s1 = 'c> '
c> print 'Hello'
Hello
c>
```

sys.path determines the interpreters's search path for modules, with the default path taken from PYTHONPATH

- Can be modified with append() (ex. *Sys.path.append('SOMEPATH')*)

The *dir()* Function

- Used to find the names a module defines and returns a sorted list of strings
  - *>>> import mod*

    >>> dir(mod)

    ['_name_', 'm1', 'm2']
- Without arguments, it lists the names currently defined (variables, modules, functions, etc)
- Does not list names of built-in functions and variables
  - Use _*bulltin*_to view all built-in functions and variables

Packages

- "dotted module names" (ex. *a.b)*
  - Submodule *b* in package *a*
- Saves authors of multi-module packages from worryi ng about each other's module names
- Python searches through *sys.path* directories for the package subdirectory
- Users of the package can import individual modules fr om the package
- Ways to import submodules
  - *import sound.effects.echo*
  - *from sound.effects import echo*
- Submodules must be referenced by full name
- An *ImportError* exception is raised when the package cannot be found

Importing * From a Package

- * does not import all submodules from a package
- Ensures that the package has been imported, only importing the names of the submodules defined in the package
- *import sound.effects.echo*

  import sound.effects.surround

  *from sound.effects import **

Intra-package References

- Submodules can refer to each other
  - *Surround* might use *echo* module
  - *import echo* also loads *surround* module
- *import* statement first looks in the containing pac kage before looking in the standard module searc h path
- Absolute imports refer to submodules of sibling p ackages
  - *sound.filters.vocoder* uses *echo* module
    from sound.effects import echo
- Can write explicit relative imports
  - *from . import echo*
  - *from .. import formats*
  - *from ..filters import equalizer*

Packages in Multiple Directories

- *_path_* is a list containing the name of the directory holding the package's *_init_.py*
- Changing this variable can affect futute searches for modules and subpackages in the package
- Can be used to extend the set of modules in a package
- Not often needed

Sources

- http://docs.python.org/tutorial/modules.html

Some Useful Packages and Resources

Useful packages:

- NumPy – similar to Numeric, but handles arrays slightly differently and has a few other built-in commands and functions
- SciPy – useful for numerical integration, ODE solutions, interpolations, etc.: based on NumPy

- Books:
  - *Learning Python*, Mark Lutz & David Ascher, O'Reilly Press, 1999
  - *Programming Python*, Mark Lutz, O'Reilly Press, 2006
  - *Core Python Programming (2nd Edition)*, Wesley J. Chun, Prentice Hall, 2006

- Websites:
  - http://docs.python.org – online version of built-in Python function documentation
  - http://rgruet.free.fr – long version of Python Quick Reference Card
  - http://mail.python.org – extensive Python forum