Introduction to Python

Lecture 2 – Fall 2021

Guido van Rossum: ZOPE

Outline

- interactive "shell"
- basic types: numbers, strings
- container types: lists, dictionaries, tuples
- variables
- control structures
- functions & procedures
- classes & instances
- modules & packages
- exceptions
- files & standard library

Interactive Shell

- Great for learning the language
- Great for experimenting with the library
- Great for testing your own modules
- Two variations: GUI,
  python (command line)
- Type statements or expressions at prompt:

```
>>> print("Hello, world“)
Hello, world
>>> x = 12**2
>>> x/2
72
>>> # this is a comment
```

Guido van Rossum: ZOPE

Standard Data Types

- Numbers
- String
- List
- Tuple
- Dictionary

Python Numbers

- Number data types store numeric values. Number objects are created when you assign a value to them. For example

  var1 = 1

- The usual suspects
  - 12, 3.14, 0xFF, (-1+2)*3/4**5, abs(x), 0<x<=5
- C-style shifting & masking
  - 1<<16, x&0xff, x|1, ~x, x^y

Strings

- "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

Lists

- Flexible arrays, *not* Lisp-like linked lists
  - a = [98, "bottles of beer", ["on", "the", "wall"]]
- Same operators as for strings
  - a+b, a*3, a[0], a[-1], a[1:], len(a)
- Item and slice assignment
  - a[0] = 98
  - del a[-1]   # -> [98, "bottles", "of", "beer"]

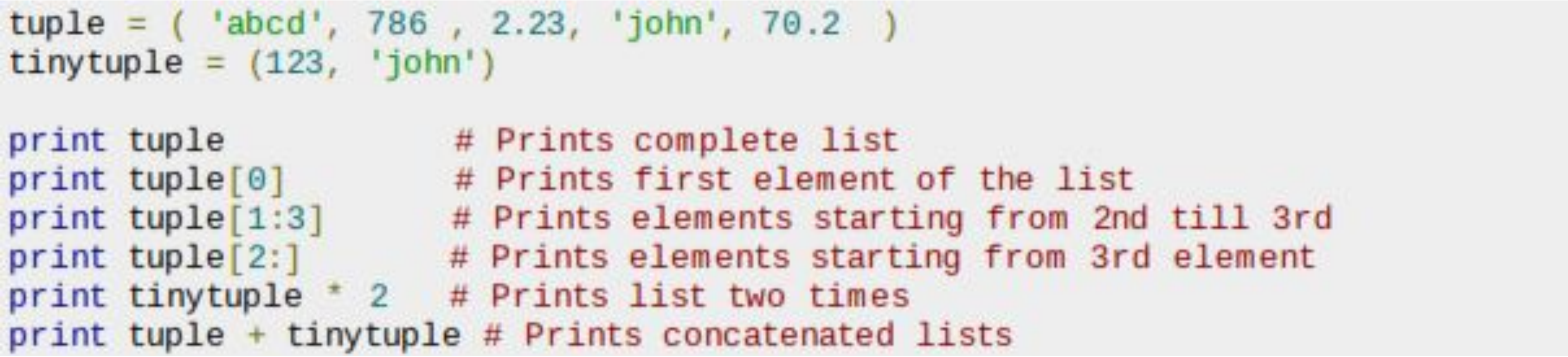
More List Operations

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

>>> a.insert(0, 42)         # [42,0,1,2,3,4]
>>> a.pop(0)           # [0,1,2,3,4]

>>> a.reverse()        # [4,3,2,1,0]
>>> a.sort()           # [0,1,2,3,4]
```

Tuples

- A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas.
- Unlike lists, however, tuples are enclosed within parentheses.
- The main differences between lists and tuples are:
  - Lists are enclosed in brackets ([ ]) and their elements and size can be changed,
  - tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as read-only lists.

Tuples



```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')

print tuple           # Prints complete list
print tuple[0]        # Prints first element of the list
print tuple[1:3]      # Prints elements starting from 2nd till 3rd
print tuple[2:]       # Prints elements starting from 3rd element
print tinytuple * 2   # Prints list two times
print tuple + tinytuple # Prints concatenated lists
```

Dictionaries

- Hash tables, "associative arrays"
  - d = {"duck": "eend", "water": "water"}
- Lookup:
  - d["duck"] -> "eend"
  - d["back"] # raises KeyError exception
- Delete, insert, overwrite:
  - del d["water"] # {"duck": "eend"}
  - d["back"] = "rug" # {"duck": "eend", "back": "rug"}
  - d["duck"] = "duik" # {"duck": "duik", "back": "rug"}

Dictionaries

- Keys, values, items:
  - d.keys() -> ["duck", "back"]
  - d.values() -> ["duik", "rug"]
  - d.items() -> [("duck","duik"), ("back","rug")]
- Presence check:
  - d.has_key("duck") -> 1; d.has_key("spam") -> 0
- Values of any type; keys almost any
  - {"name":"Guido", "age":43, ("hello","world"):1, 42:"yes", "flag": ["red","white","blue"]}

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
```

- ***Everything*** is a "variable":
  - Even functions, classes, modules

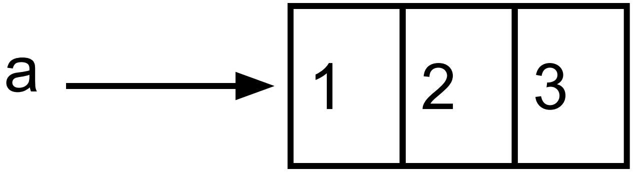
Reference Semantics

- 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:

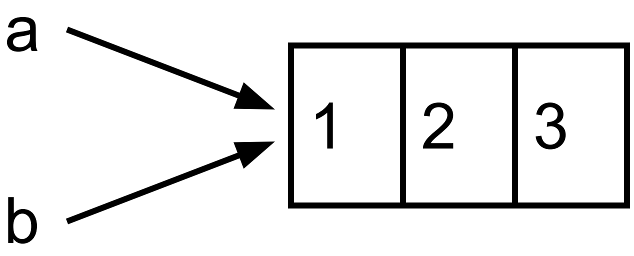
```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> print b
[1, 2, 3, 4]
```

Changing a Shared List

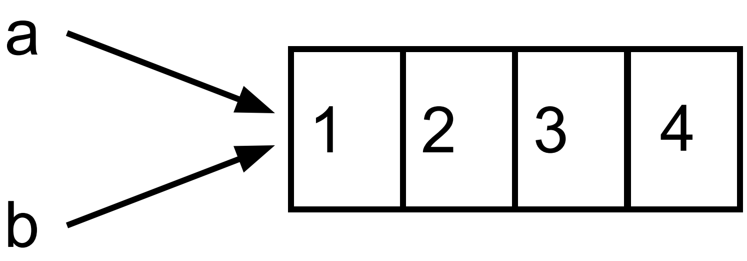
```
a = [1, 2, 3]
```



```
b = a
```

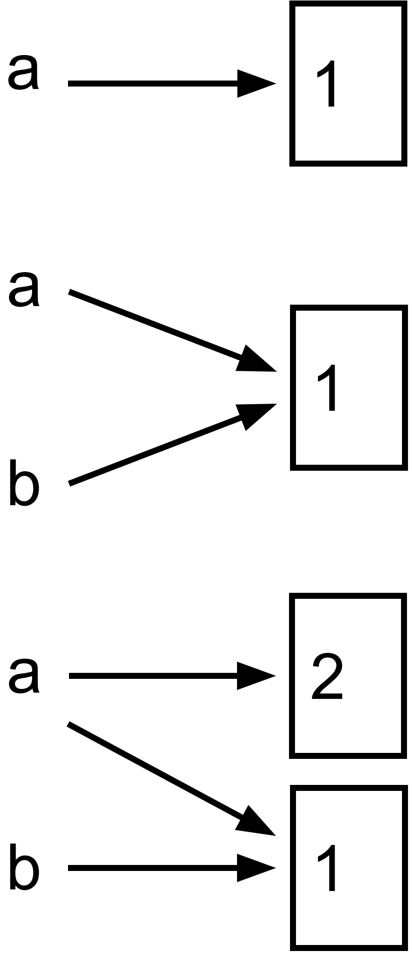


```
a.append(4)
```



Changing an Integer

```
a = 1

b = a

a = a+1
```



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

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

Control Structures

```
if condition:
    statements
[elif condition:
    statements] ...
else:
    statements
```

```
while condition:
    statements

for var in sequence:
    statements

break
continue
```

Grouping Indentation

In Python:

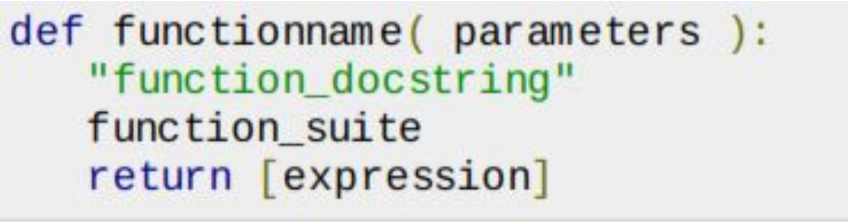
```
for i in range(20):
    if i%3 == 0:
        print i
        if i%5 == 0:
            print("Bingo!“)
    print("---“)
```

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");
}
```

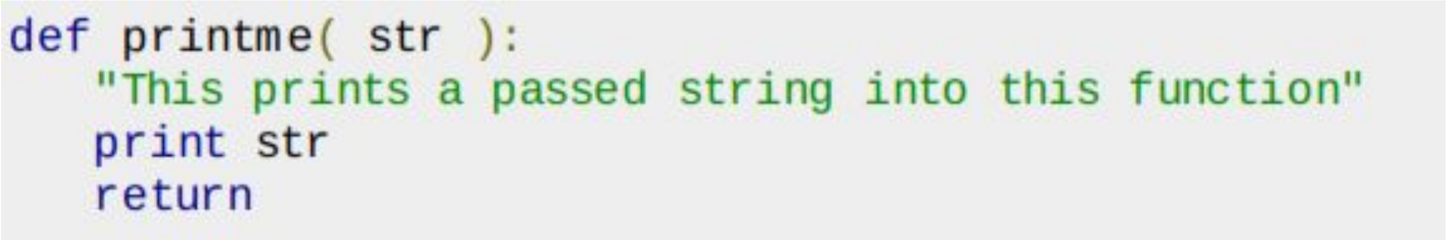
Functions

- Syntax



```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

- Example



```
def printme( str ):
   "This prints a passed string into this function"
   print str
   return
```

Functions

- Calling a Function: printme(“hello”)
- Function Arguments
  - Required arguments
  - Keyword arguments
  - Default arguments
  - Variable-length arguments
- return statement

https://www.tutorialspoint.com/python/python_functions.htm

Functions

- Scope of Variable
  - Global
  - Local

Classes

```
class name:
    "documentation"
    statements
```

-or-

```
class name(base1, base2, ...):
    ...
```

Most, *statements* are method definitions:

```
    def name(self, arg1, arg2, ...):
        ...
```

May also be *class variable* assignments

Example Class

```
class Stack:
    "A well-known data structure…"
    def __init__(self):          # constructor
        self.items = []
    def push(self, x):
        self.items.append(x)   # the sky is the limit
    def pop(self):
        x = self.items[-1]       # what happens if it’s empty?
        del self.items[-1]
        return x
    def empty(self):
        return len(self.items) == 0# Boolean result
```

Using Classes

- To create an instance, simply call the class object:

```
x = Stack()     # no 'new' operator!
```

- To use methods of the instance, call using dot notation:

```
x.empty()        # -> 1
x.push(1)                        # [1]
x.empty()        # -> 0
x.push("hello")                  # [1, "hello"]
x.pop()          # -> "hello"      # [1]
```

- To inspect instance variables, use dot notation:

```
x.items          # -> [1]
```

Class/ instance Variables

```
class Connection:
    verbose = 0                          # class variable
    def __init__(self, host):
        self.host = host                     # instance variable
    def debug(self, v):
        self.verbose = v
    def connect(self):
        if self.verbose:                 # class or instance variable?
            print(“hello”)
```

instance Variable Rules

- On use via instance (self.x), search order:
  - (1) instance, (2) class, (3) base classes
  - this also works for method lookup
- On assignment via instance (self.x = ...):
  - always makes an instance variable
- More
  - mutable *class* variable: one copy *shared* by all
  - mutable *instance* variable: each instance its own

Modules

- Collection of stuff in *foo*.py file
  - functions, classes, variables
- Importing modules:
  - import re; print re.match("[a-z]+", s)
  - from re import match; print match("[a-z]+", s)
- Import with rename:
  - import re as regex
  - from re import match as m

Packages

- Collection of modules in directory
- May contain subpackages
- Import syntax:
  - from P.Q.M import foo; print foo()
  - from P.Q import M; print M.foo()
  - import P.Q.M; print P.Q.M.foo()
  - import P.Q.M as M; print M.foo() # new

Exceptions

```
def foo(x):
    return 1/x

def bar(x):
    try:
        print foo(x)
    except ZeroDivisionError, message:
        print "Can’t divide by zero:", message

bar(0)
```

Exceptions: Try - finally

```
f = open(file)
try:
    process_file(f)
finally:
    f.close() # always executed
print "OK" # executed on success only
```

Exceptions: Try - finally

```
f = open(file)
try:
    process_file(f)
finally:
    f.close() # always executed
print "OK" # executed on success only
```

Raising Exceptions

- raise IndexError
- raise IndexError("k out of range")
- raise IndexError, "k out of range"
- try:

```
try:
    something
except:     # catch everything
    print "Oops"
    raise    # reraise
```

□ More about Python Operators:

https://www.programiz.com/python-programming/operators