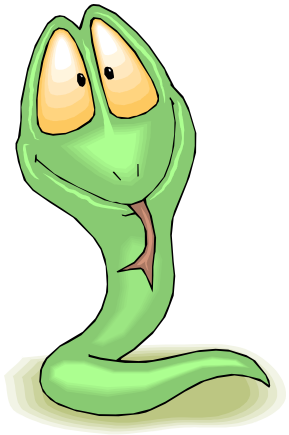
Python: A Simple Introduction

part i

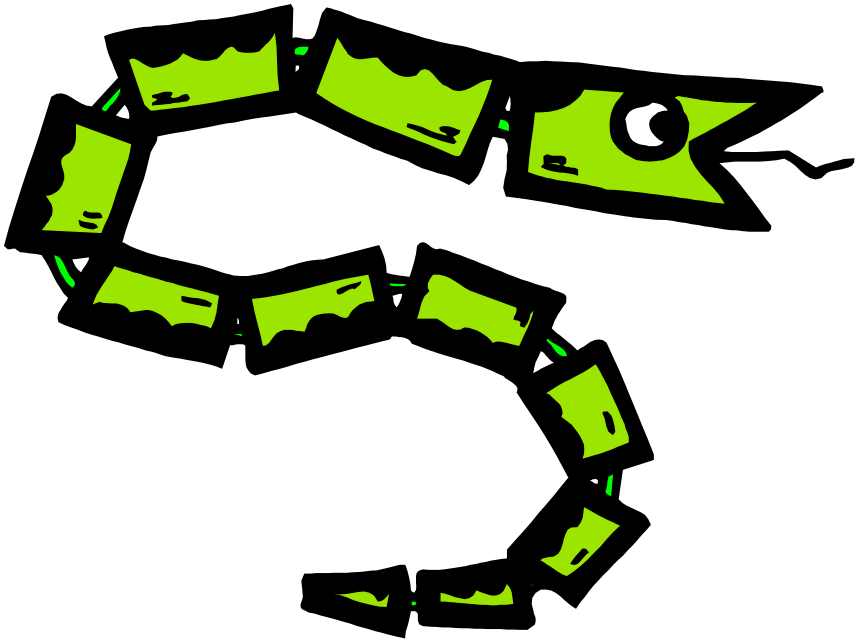


Slides adapted from Mitch Marcus

Contents

- The basics
- Logical Expressions
- Sequence Types: Tuples, Lists, and Strings
- Mutability: Tuples vs. Lists
- Flow Control
- Dictionaries
- Functions in Python
- Importing and Modules

The Basics



A Simple Example Script

```
x = 34 - 23                 # A comment.
y = "Hello"                 # Another one.
z = 3.45
if z == 3.45 or y == "Hello":
     x = x + 1
     y = y + " World"   # String concatenation
print x
print y
```

Basic Data Types

- **Integers (default for numbers)**

```
z = 5 / 2    # Answer is 2, integer division.
```

- **Floats**

```
x = 3.456
x = float(3)
```
or `x = 3.`

- **Strings**

- Can use " " or ' ' to specify.

`"abc"  'abc'` (same thing.)

- Unmatched quotes can occur within the string.

`"matt's"` or `'matt"s'`

- Use triple double-quotes for multi-line strings or strings that contain both ' and " :

`"""a'b"c"""`

Python and Types

Python determines the data types of *variable bindings* in a program automatically. *"Dynamic Typing"*

But Python's not casual about types, it enforces the types of *objects*. *"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 bound to a string.
y = 23                # Decides y is bound to an integer.
 print x + y   # Python will complain about this.
```

Whitespace

Whitespace is meaningful in Python: especially indentation and placement of newlines.

- **Use a newline to end a line of code.**
- Use `\` when must go to next line prematurely.
- **No braces `{ }` to mark blocks of code in Python… Use *consistent* indentation instead.**
- The first line with *less* indentation is ending the block
- The first line with *more* indentation starts a nested block
- **Often a colon appears at the start of a new block. (e.g. for function and class definitions)**

A Simple Example Script

```
x = 34 - 23                 # A comment.
y = "Hello"                 # Another one.
z = 3.45
 if z == 3.45 or y == "Hello":
     x = x + 1
     y = y + " World"    # String concatenation
 print x
 print y
```

Comments

- Start comments with # – the rest of line is ignored.
- Can include a "documentation string" as the first line of any new function or class that you define.

```
def my_function(x, y):
    """This is the docstring. This
       function does blah blah blah."""
    # The code would go here...
```

- The development environment, debugger, and other tools use it: it's good style to include one.

Accessing Non-Existent Names

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

```
>>> y

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    y
NameError: name 'y' is not defined
>>> y = 3
>>> y
3
```

Multiple Assignment

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

```
>>> x = y = 1
>>> x
1
>>> y
1
```

and

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

Naming Rules

- 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:
- 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

Logical Expressions

True and False

- **Other values equivalent to `True` and `False`:**

  `False`: zero,

  `True`: non-zero numbers

  (also: non-empty objects: “if [1]” evaluates to True]

- **Comparison operators: `==`, `!=`, `<`, `<=`, `>`, `>=`**

  X and Y have same value: **`X == Y`**

  Compare with **`X is Y`** :

  —X and Y are two variables that refer to *identical objects.*

Boolean Logic Expressions

- **You can also combine Boolean expressions.**
  - *True* if a is true and b is true: `a and b`
  - *True* if a is true or b is true: `a or b`
  - *True* if a is false: `not a`
- **Use parentheses as needed to disambiguate complex Boolean expressions.**

Conditional Expressions

```
x = true_value if condition else false_value
```

- **First, `condition` is evaluated**

If True, `true_value` is evaluated and returned

If False, `false_value` is evaluated and returned

Sequence types:
Tuples, Lists, and Strings

Sequence Types

- Tuple
- ***Immutable*** ordered sequence of items
- Items can be of mixed types

- Strings
- ***Immutable***
- Conceptionally very much like a tuple

- List
- ***Mutable*** ordered sequence of items
- Items can be of mixed types

Similar Syntax

- **All three sequence types (tuples, strings and lists) share much of their syntax and functionality.**

- The operations shown in this section can be applied to *all* sequence types
- **most examples will just show the operation performed on one**

Sequence Types 1

- Tuples are defined using parentheses (and commas).

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Lists are defined using square brackets (and commas).

```
>>> li = ["abc", 34, 4.34, 23]
```

- Strings are defined using quotes (", ', or """).

```
>>> st = "Hello World"
>>> st = 'Hello World'
>>> st = """This is a multi-line
string that uses triple quotes."""
```

Sequence Types 2

- **We can access individual members of a tuple, list, or string using square bracket “array” notation.**
- ***Note that all are 0 based…***

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]      # Second item in the tuple.
'abc'

>>> li = ["abc", 34, 4.34, 23]
>>> li[1]       # Second item in the list.
34

>>> st = "Hello World"
>>> st[1]    # Second character in string.
'e'
```

Positive and negative indices

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0.

```
>>> t[1]
'abc'
```

Negative lookup: count from right, starting with –1.

```
>>> t[-3]
4.56
```

Slicing: Return Copy of a Subset 1

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying *before* the second index.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

You can also use negative indices when slicing.

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

Slicing: Return Copy of a Subset 2

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Omit the first index to make a copy starting from the beginning of the container.

```
>>> t[:2]
(23, 'abc')
```

Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> t[2:]
(4.56, (2,3), 'def')
```

Copying the Whole Sequence

To make a *copy* of an entire sequence, you can use `[:]`.

```
>>> t[:]
(23, 'abc', 4.56, (2,3), 'def')
```

Note the difference between these two lines for mutable sequences:

```
>>> list2 = list1 # 2 names refer to 1 ref
    # Changing one affects both

>>> list2 = list1[:] # Two independent copies, two refs
```

The ‘in’ Operator

- Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- For strings, tests for substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

The + Operator

- The + operator produces a *new* tuple, list, or string whose value is the concatenation of its arguments.

```
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)

>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]

>>> "Hello" + " " + "World"
'Hello World'
```

The * Operator

- **The * operator produces a *new* tuple, list, or string that "repeats" the original content.**

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

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> "Hello" * 3
'HelloHelloHello'
```

Mutability: Tuples vs. Lists

Tuples: Immutable

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14

Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item assignment
```

You **can't change a tuple**.

You can make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```

Lists: Mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
   ['abc', 45, 4.34, 23]
```

- We can change lists *in place.*
- Name *li* still points to the same memory reference when we're done.
- The mutability of lists means that they aren't as fast as tuples.

Operations on Lists Only 1

```
>>> li = [1, 11, 3, 4, 5]

>>> li.append('a')  # Our first exposure to method
   syntax
>>> li
[1, 11, 3, 4, 5, 'a']

>>> li.insert(2, 'i')
>>> li
[1, 11, 'i', 3, 4, 5, 'a']
```

The *extend* method vs the + operator.

- `+` creates a fresh list (with a new memory reference => also valid for tuples)
- *extend* operates on list `li` in place.

```
>>> li.extend([9, 8, 7])
>>>li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

Maybe confusing:

Extend takes a list as an argument.

Append takes a singleton as an argument.

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

Operations on Lists Only 3

```
>>> li = ['a', 'b', 'c', 'b']

>>> li.index('b')      # index of first occurrence, also for tuples
1

>>> li.count('b')      # number of occurrences, also for tuples
2

>>> li.remove('b')     # remove first occurrence
>>> li
  ['a', 'c', 'b']
```

Operations on Lists Only 4

```
>>> li = [5, 2, 6, 8]

>>> li.reverse()     # reverse the list *in place*
>>> li
  [8, 6, 2, 5]

>>> li.sort()        # sort the list *in place*
>>> li
  [2, 5, 6, 8]
```

Tuples vs. Lists

- **Lists** are **slower** but **more powerful** than tuples.
- **Lists** can **be modified**, and they have **lots of handy operations** we can perform on them.
- **Tuples** are **immutable** and have fewer features.

- To **convert** between tuples and lists use the list() and tuple() functions:

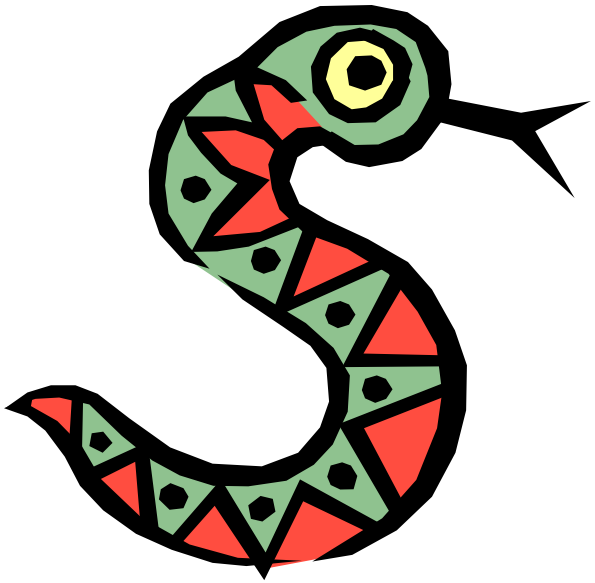
```
li = list(tu)
tu = tuple(li)
```

Remark:

we can use same semantics to convert any datatype

`a = int (3./10.)` results in an int with value 0

Flow Control



Flow Control

- **There are several Python expressions that control the flow of a program. All of them make use of boolean conditional tests.**
- *if* statements
- *while* and *for* loops
- *assert*, *break*, *continue* statements

if Statements

```
if x == 3:
    print "X equals 3."
elif x == 2:
    print "X equals 2."
else:
    print "X equals something else."
print "This is outside the 'if'."
```

Note:

- **Use of indentation for blocks**
- **Colon (*:*) before block starts**

while Loops

```
x = 3
while x < 10:
    x = x + 1
    print "Still in the loop."
print "Outside the loop."
```

break and *continue* and *assert*

- **Use *break* inside a loop to leave it before stopping condition is fulfilled.**

- **You can use the keyword *continue* inside a loop to stop processing the current iteration of the loop and to immediately go on to the next one.**
-

- **An *assert([condition])* statement will stop the program if its argument is false.**

For Loops 1

- A for-loop steps through each of the items in a list, tuple, string, or any other type of object which is “iterable”

```
for <item> in <collection>:
  <statements>
```

- If <collection> is a list or a tuple, then the loop steps through each element of the sequence.

- If <collection> is a string, then the loop steps through each character of the string.

```
for someChar in “Hello World”:
   print someChar
```

For Loops 2

```
for <item> in <collection>:
  <statements>
```

- **<item> can be more complex than a single variable name.**
- When the elements of <collection> are themselves sequences, then <item> can match the structure of the elements.

- This multiple assignment can make it easier to access the individual parts of each element.

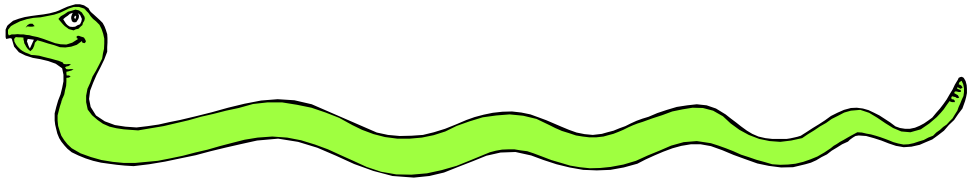
```
for (x, y) in [(a,1), (b,2), (c,3), (d,4)]:
  print x
```

For loops and the *range()* function

- **range(5) returns [0,1,2,3,4]**
- **So we could say:**

```
for x in range(5):
    print x
```

Dictionaries



Dictionaries: A *Mapping* type

- **Dictionaries store a *mapping* between a set of keys and a set of values.**
  - Keys can be any *immutable* type.
  - Values can be any type
  - A single dictionary can store values of different types
- **You can define, modify, view, lookup, and delete the key-value pairs in the dictionary.**

Creating and accessing dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user']
'bozo'

>>> d['pswd']
1234

>>> d['bozo']

Traceback (innermost last):
  File '<interactive input>' line 1, in ?
KeyError: bozo
```

Updating Dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user'] = 'clown'
>>> d
{'user':'clown', 'pswd':1234}
```

- Keys must be unique.
- Assigning to an existing key replaces its value.

```
>>> d['id'] = 45
>>> d
{'user':'clown', 'id':45, 'pswd':1234}
```

- Dictionaries are unordered
- New entry might appear anywhere in the output.

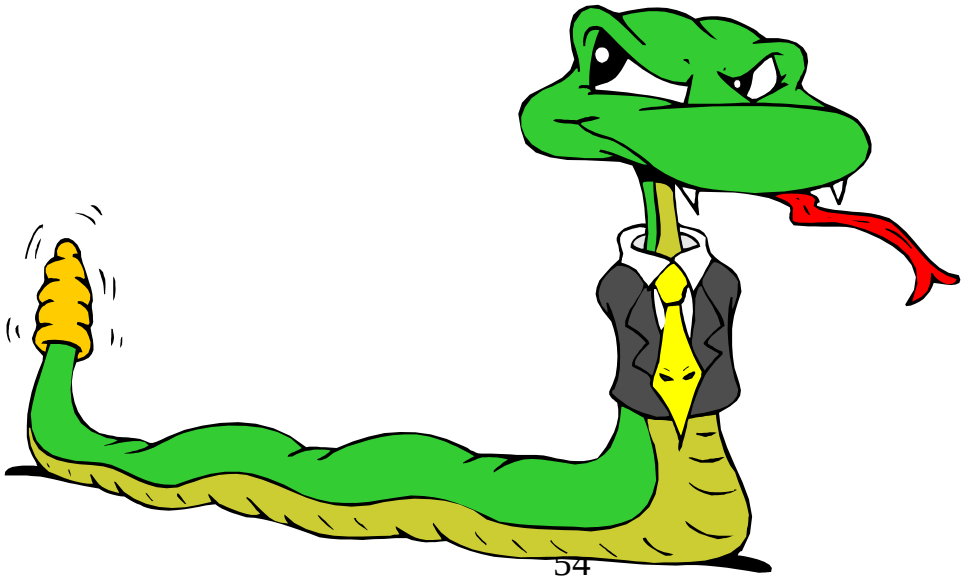
Removing dictionary entries

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> del d['user']                  # Remove one.
>>> d
{'p':1234, 'i':34}

>>> d.clear()                      # Remove all.
>>> d
{}
```

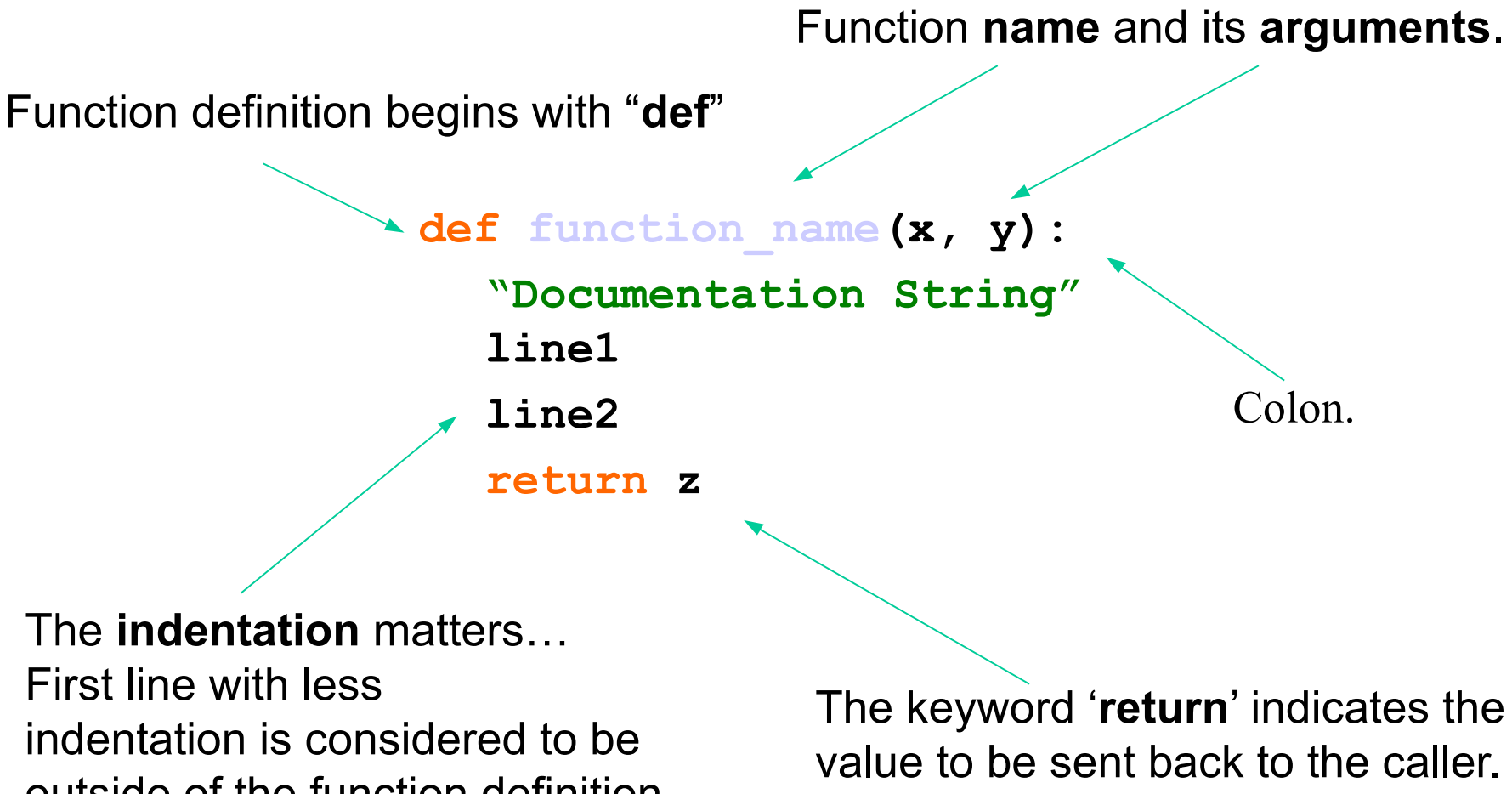
Useful Accessor Methods

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> d.keys()                # List of keys.
['user', 'p', 'i']

>>> d.values()              # List of values.
['bozo', 1234, 34]

>>> d.items()               # List of item tuples.
[('user','bozo'), ('p',1234), ('i',34)]
```

Functions in Python



Defining Functions



Calling a function

- The **syntax** for a **function call** is:

```
>>> def myfun(x, y):
        return x * y
>>> myfun(3, 4)
12
```

- Functions are specified by their **name**
- They can pass their functionality (change name)

```
>>> f = myfun
>>> myfun = 10
>>> f(3, 4)
  12
```

The documentation string

Functions can be accompanied with a documentation:

```
>>> def squareRoot(x):
"Returns the square root of x"
return math.sqrt(x)

>>> squareRoot(10)
100

>>> help( squareRoot )
square(x)
Returns the square root of x
```

Useful function statement

`>>> assert(x > 0),` makes sure that the argument is positive

Calling a Function with default values

- Functions can be setup with **default values**:

```
>>> def myfun(x=1., y=5.):
        return x * y

>>> myfun()
5.
>>> myfun(5.)
25.
>>> myfun(y=2.)
2.
```

Functions without returns

- ***All* functions in Python have a return value**
  - even if no *return* line inside the code.
- **Functions without a *return* return the special value *None*.**
  - *None* is a special constant in the language.
  - The interpreter doesn't print *None*

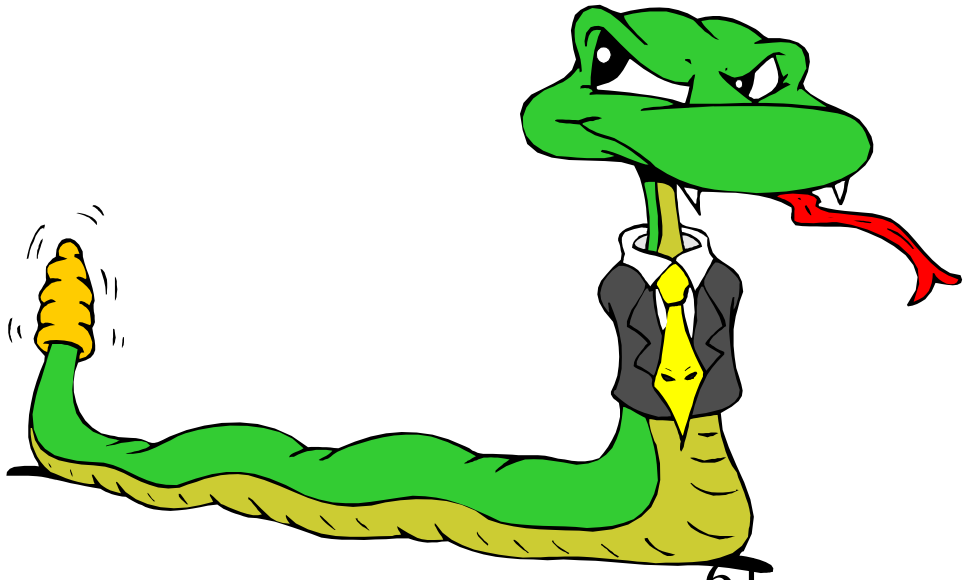
Functions are first-class objects in Python

Functions can be used as any other data type.
They can be:

- **Arguments** to function
- **Return** values of **functions**
- **Assigned** to variables,
- **Parts of** tuples, lists
- **…**

```
>>> def myfun(x):
        return x*3

  >>> def applier(q, x):
        return q(x)

  >>> applier(myfun, 7)
  21
```

Importing and Modules



Importing and Modules

- Use classes & functions defined in another file.
- A Python module is a file with the same name (plus the *.py* extension)
- Like Java *import*, C++ *include*.
- Three formats of the command:

```
import somefile (as sf)
from somefile import *
from somefile import className
```

import …

```
import somefile
```

- ***Everything*** **in somefile.py gets imported.**
- **To refer to something in the file, append the text "somefile." to the front of its name:**

```
somefile.myFunction(34)
somefile.className.method("abc")
```

```
import somefile as sf
```

- **To refer to something in the file, append the text "sf." to the front of its name.**

*from … import **

```
from somefile import *
```

- ***Everything*** **in somefile.py gets imported**
- **To refer to anything in the module, just use its name. Everything in the module is now in the current namespace.**
- ***Caveat!*** **Using this import command can easily overwrite the definition of an existing function or variable!**

```
myFunction(34)

className.method("abc")
```

from … import …

```
from somefile import myFunction
```

- **Only the item *myFunction* in somefile.py gets imported.**
- **After importing *myFunction*, you can just use it without a module prefix. It's brought into the current namespace.**
- ***Caveat*! This will overwrite the definition of this particular name if it is already defined in the current namespace!**

```
myFunction(34)
```
 This got imported by this command.

```
myOtherFunction(34)
```
 This one didn't.

Commonly Used Modules

- **Some useful modules to import coming along with Python installation:**

- **Module: sys - Lots of handy stuff.**
- argv, …
- **Module: os - OS specific code.**
- listdir, system, …
- **Module: os.path - Directory processing.**
- exists, …

More Commonly Used Modules

- **Module: math - Mathematical code.**
- sin, cos, exp, log, sqrt, …

- **Module: Random - Random number code.**
- uniform, choice, shuffle

For Scientists

- **Module: numpy - basis for numerics**
- powerful numpy array
- efficient functions on numpy arrays
- **Module: scipy - advanced methods**
- linear algebra
- integration, ODE solving, …
- **Module: matplotlib - matlab like plotting**
- plotting

Finally: Getting help

- **Interactive help**
- help()

- **Information on functions**
- help(‘print’) shows help for print statement
- help(‘os’) shows help on module OS
- help(‘os.system’) shows help on command system of module OS
- help([1,2]) shows help on list object

- **Very useful command**
- dir() shows all stuff that was defined in your actual session
- **Useful auto-completion with <tab>**
- random. <tab> shows all functions, classes and so on

- **Documentation on the web:**

http://docs.python.org

Have fun solving some exercises!