Thực hành Python

Nguyễn Mạnh Hiển
hiennm@tlu.edu.vn

Cài đặt Python

Kiểm tra xem máy tính của bạn đã có Python chưa?

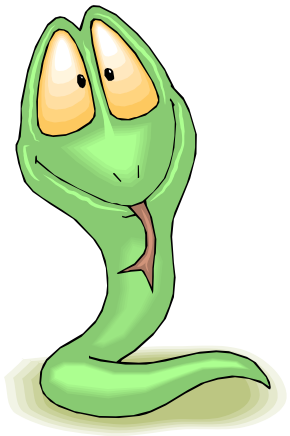
- Nếu đã có **Python chuẩn**, mở **IDLE** ra để thực hành.
- Nếu đã có **Anaconda** (= Python chuẩn + các thư viện bổ sung), mở **Spyder** ra để thực hành.
- Nếu máy tính của bạn chưa có Python:
  1. Tải xuống bộ cài Python chuẩn ở đây: https://www.python.org/downloads/
  2. Tiến hành cài đặt theo chỉ dẫn trên màn hình.
  3. Mở IDLE ra để thực hành.

Soạn thảo và chạy chương trình Python

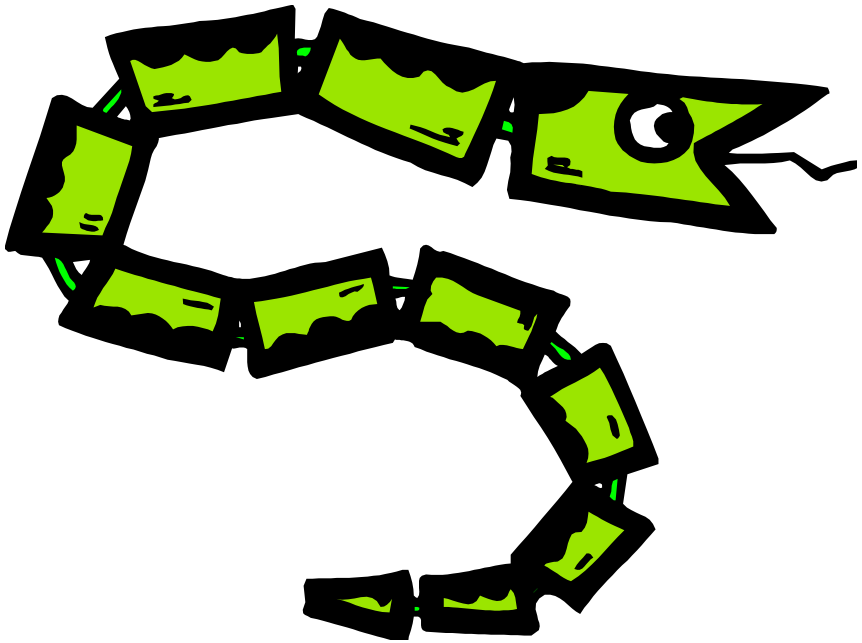
- **Cách 1:** Gõ lệnh trực tiếp ở dấu nhắc lệnh rồi nhấn Enter để chạy.
- **Cách 2:**
  - Trong IDLE:
    - Chọn menu File → New File để mở cửa sổ soạn thảo.
    - Gõ vào chương trình rồi nhấn F5 để chạy.
  - Trong Spyder:
    - Đã có sẵn cửa sổ soạn thảo.
    - Gõ vào chương trình rồi nhấn F5 để chạy.
- Trong các slide hướng dẫn theo sau, phải thêm cặp ngoặc để sửa lệnh "**`print something`**" thành "**`print(something)`**".
  - Ta sẽ dùng Python 3 để thực hành, nhưng các slide đó đang dùng Python 2.

Python: A Simple Tutorial

Slides by Matt Huenerfauth



The Basics



Our Code Sample in IDLE

```
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 concat.
print x
print y
```

Chú ý khi copy chương trình ở đây vào IDLE/Spyder thì phải gõ lại các dấu nháy kép và dấu trừ.

Enough to Understand the Code

- **Assignment uses `=` and comparison uses `==`.**
- **For numbers `+ - * / %` are as expected.**
  - Special use of `+` for string concatenation.
  - Special use of `%` for string formatting (as with printf in C)
- **Logical operators are words (`and, or, not`)**
  ***not* symbols**
- **The basic printing command is `print`.**
- **The first assignment to a variable creates it.**
  - Variable types don't need to be declared.
  - Python figures out the variable types on its own.

Chú ý trong Python 3 thì dấu "/" là chia thông thường.
Muốn chia lấy phần nguyên thì dùng "//".

Basic Datatypes

- **Integers (default for numbers)**

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

- **Floats**

```
x = 3.456
```

- **Strings**
  - Can use “” or ‘’ to specify.
    “abc” ‘abc’ (Same thing.)
  - Unmatched can occur within the string.
    “matt’s”
  - Use triple double-quotes for multi-line strings or strings than contain both ‘ and “ inside of them:
    “““a‘b“c”””

Chú ý trong Python 3 thì dấu "/" là chia thông thường.
Muốn chia lấy phần nguyên thì dùng "//".

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 outside of 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.)**

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

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

Assignment

- ***Binding a variable*** **in Python means setting a *name* to hold a *reference* to some *object*.**
  - *Assignment creates references, not copies*

- **Names in Python do not have an intrinsic type. Objects have types.**
  - Python determines the type of the reference automatically based on the data object assigned to it.

- **You create a name the first time it appears on the left side of an assignment expression:**

```
x = 3
```

- **A reference is deleted via garbage collection after any names bound to it have passed out of scope.**

Accessing Non-Existent Names

- 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 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 = 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
```

Sequence types:
Tuples, Lists, and Strings

Sequence Types

1. Tuple
   - A simple ***immutable*** ordered sequence of items
   - Items can be of mixed types, including collection types

2. Strings
   - ***Immutable***
   - **Conceptually very much like a tuple**

3. List
   - ***Mutable*** ordered sequence of items of mixed types

Similar Syntax

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

- Key difference:
  - **Tuples and strings are *immutable***
  - **Lists are *mutable***
- 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
```

- **Be careful: the *in* keyword is also used in the syntax of *for loops* and *list comprehensions*.**

The + Operator

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

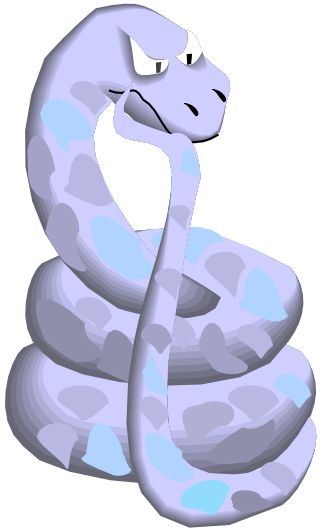
```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)

>>> [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)**
- ***extend* operates on list `li` in place.**

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

***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
1

>>> li.count('b')      # number of occurrences
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]

>>> li.sort(some_function)
    # sort in place using user-defined comparison
```

Tuples vs. Lists

- **Lists 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)
```

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.
- (Dictionaries work by *hashing*)

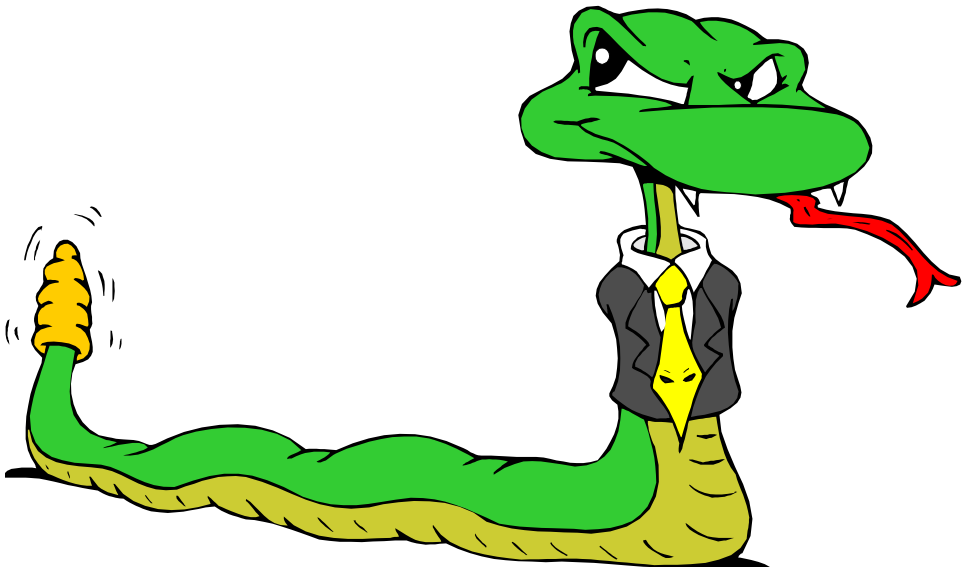
Removing dictionary entries

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

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

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

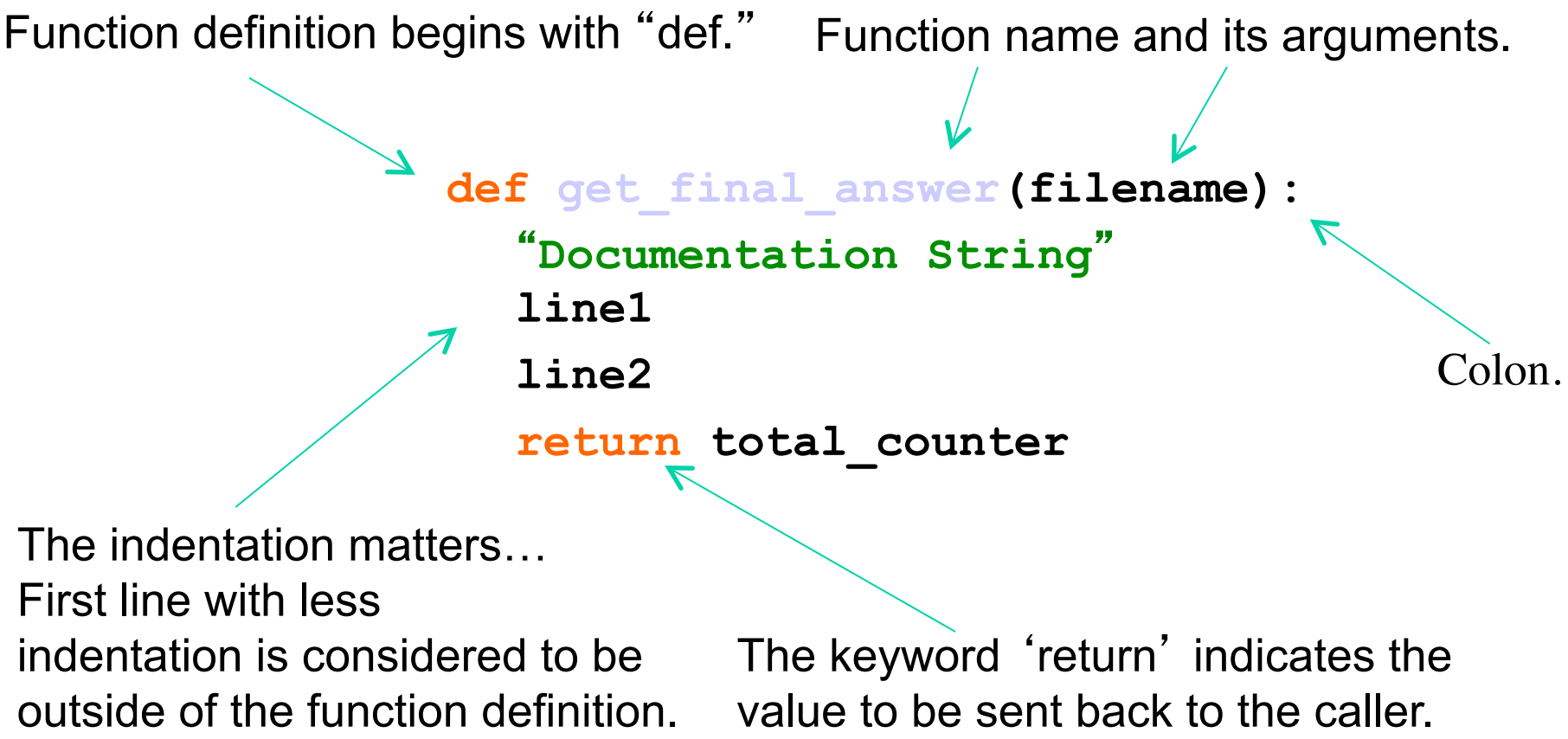
Useful Accessor Methods

```
>>> 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



No header file or declaration of types of function or arguments.

Calling a Function

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

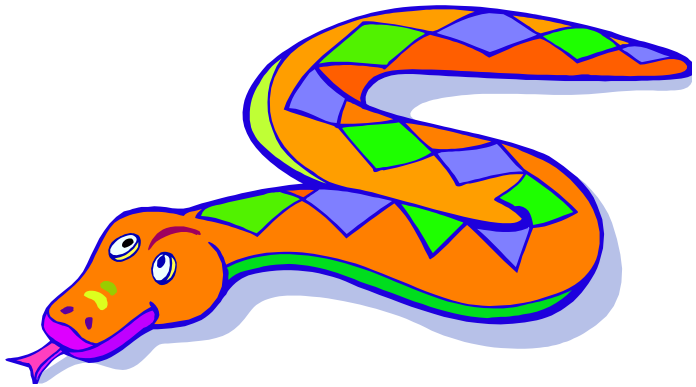
```
>>> def myfun(x, y):
        return x * y
>>> myfun(3, 4)
12
```

- **Parameters in Python are "Call by Assignment."**
  - Sometimes acts like "call by reference" and sometimes like "call by value" in C++.
    - Mutable datatypes: Call by reference.
    - Immutable datatypes: Call by value.

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.
  - *None* is used like *NULL*, *void*, or *nil* in other languages.
  - *None* is also logically equivalent to False.
  - The interpreter doesn't print *None*

Logical Expressions



True and False

- ***True*** **and** ***False*** **are constants in Python.**

- **Other values equivalent to** ***True*** **and** ***False*****:**
  - *False*: zero, *None*, empty container or object
  - *True*: non-zero numbers, non-empty objects

- **Comparison operators: ==, !=, <, <=, etc.**
  - X and Y have same value: `X == Y`
  - Compare with `X is Y` :
    - —X and Y are two variables that refer to the *identical same object.*

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.**

Special Properties of *and* and *or*

- **Actually *and* and *or* *don't* return *True* or *False*.**
- **They return the value of one of their sub-expressions (which may be a non-Boolean value).**
- `X and Y and Z`
  - If all are true, returns value of Z.
  - Otherwise, returns value of first false sub-expression.
- `X or Y or Z`
  - If all are false, returns value of Z.
  - Otherwise, returns value of first true sub-expression.
- ***And* and *or* use *lazy evaluation*, so no further expressions are evaluated**

The "and-or" Trick

- **A trick to implement a simple conditional**

```
result = test and expr1 or expr2
```

  - When test is *True*, result is assigned expr1.
  - When test is *False*, result is assigned expr2.
  - Works almost like `(test ? expr1 : expr2)` expression of C++.

- ***But* if the value of expr1 is *ever False,* the trick doesn't work.**
- **Avoid (hard to debug), but you may see it in the code.**
- **Made unnecessary by conditional expressions in Python 2.5 (see next slide)**

Conditional Expressions: New in Python 2.5

- `x = true_value if condition else false_value`
- Uses lazy evaluation:
  - **First, `condition` is evaluated**
  - **If *True*, `true_value` is evaluated and returned**
  - **If *False*, `false_value` is evaluated and returned**
- Suggested use:
- `x = (true_value if condition else false_value)`

Control of Flow

Control of Flow

- **There are several Python expressions that control the flow of a program. All of them make use of Boolean conditional tests.**
  - *if* Statements
  - *while* Loops
  - *assert* 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'."
```

Be careful! The keyword *if* is also used in the syntax of filtered *list comprehensions*.

Note:

- **Use of indentation for blocks**
- **Colon (*:*) after boolean expression**

while Loops

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

break and *continue*

- You can use the keyword *break* inside a loop to leave the *while* loop entirely.

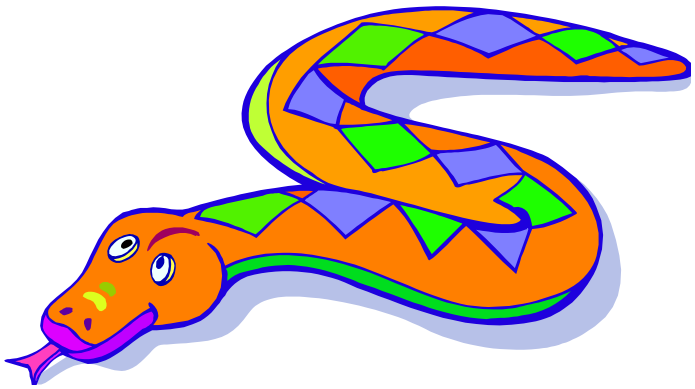
- You can use the keyword *continue* inside a loop to stop processing the current iteration of the loop and to immediately go on to the next one.

assert

- **An *assert* statement will check to make sure that something is true during the course of a program.**
  - If the condition if false, the program stops.

```
assert(number_of_players < 5)
```

Generating Lists using “List Comprehensions”



List Comprehensions

- **A powerful feature of the Python language.**
  - Generate a new list by applying a function to every member of an original list.
  - Python programmers use list comprehensions extensively. You'll see many of them in real code.

- **The syntax of a *list comprehension* is somewhat tricky.**
  - Syntax suggests that of a *for*-loop, an *in* operation, or an *if* statement
    - —all three of these keywords ('*for*', '*in*', and '*if*') are also used in the syntax of forms of list comprehensions.

Using List Comprehensions 1

```
>>> li = [3, 6, 2, 7]
>>> [elem*2 for elem in li]
[6, 12, 4, 14]
```

Note: Non-standard colors on next several slides to help clarify the list comprehension syntax.

[expression `for` name `in` list]

- **Where expression is some calculation or operation acting upon the variable name.**
- **For each member of the list, the list comprehension**
  1. **sets name equal to that member,**
  2. **calculates a new value using expression,**
- **It then collects these new values into a list which is the return value of the list comprehension.**

Using List Comprehensions 2

`[ expression for name in list ]`

- If list contains elements of different types, then expression must operate correctly on the types of all of list members.
- If the elements of list are other containers, then the name can consist of a container of names that match the type and "shape" of the list members.

```
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [ n * 3 for (x, n) in li]
[3, 6, 21]
```

Using List Comprehensions 3

[expression for name in list]

- expression can also contain user-defined functions.

```
>>> def subtract(a, b):
        return a - b

>>> oplist = [(6, 3), (1, 7), (5, 5)]
>>> [subtract(y, x) for (x, y) in oplist]
[-3, 6, 0]
```

Filtered List Comprehension 1

```
[expression for name in list if filter]
```

- **Filter determines whether expression is performed on each member of the list.**
- **For each element of list, checks if it satisfies the filter condition.**
- **If it returns *False* for the filter condition, it is omitted from the list before the list comprehension is evaluated.**

Filtered List Comprehension 2

```
[ expression for name in list if filter ]
```

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem * 2 for elem in li if elem > 4]
[12, 14, 18]
```

- Only 6, 7, and 9 satisfy the filter condition.
- So, only 12, 14, and 18 are produced.

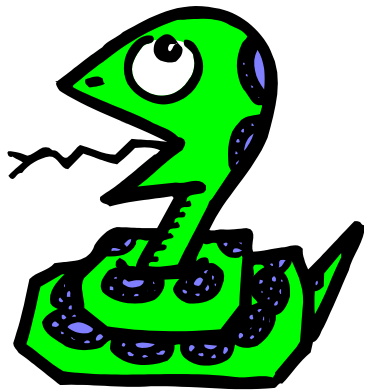
Nested List Comprehensions

- **Since list comprehensions take a list as input and produce a list as output, they are easily nested:**

```
>>> li = [3, 2, 4, 1]
>>> [elem*2 for elem in
          [item+1 for item in li] ]
[8, 6, 10, 4]
```

- **The inner comprehension produces: [4, 3, 5, 2].**
- **So, the outer one produces: [8, 6, 10, 4].**

For Loops



For Loops 1

Note: Non-standard colors on these slides.

- **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

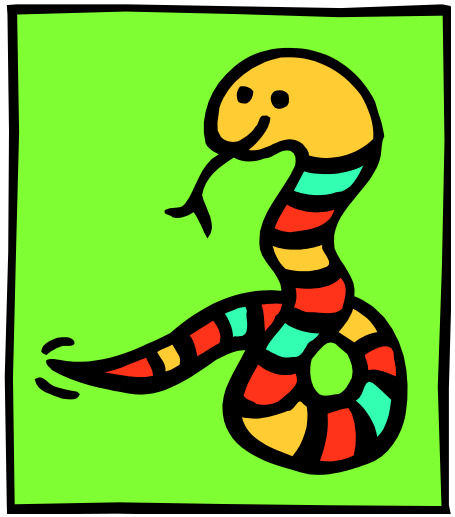
- **Since a variable often ranges over some sequence of numbers, the *range()* function returns a list of numbers from 0 up to but not including the number we pass to it.**

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

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

- **(There are more complex forms of *range()* that provide richer functionality…)**

String Operations



String Operations

- A number of methods for the string class perform useful formatting operations:

```
>>> "hello".upper()
'HELLO'
```

- Check the Python documentation for many other handy string operations.

- Helpful hint: use `<string>.strip()` to strip off final newlines from lines read from files

String Formatting Operator: *%*

- **The operator *%* allows strings to be built out of many data items in a "fill in the blanks" fashion.**
  - Allows control of how the final string output will appear.
  - For example, we could force a number to display with a specific number of digits after the decimal point.

- **Very similar to the sprintf command of C.**

```
>>> x = "abc"
>>> y = 34
>>> "%s xyz %d" % (x, y)
'abc xyz 34'
```

- **The tuple following the *%* operator is used to fill in the blanks in the original string marked with *%s* or *%d*.**
  - **Check Python documentation for whether to use %s, %d, or some other formatting code inside the string.**

Printing with Python

- You can print a string to the screen using "print."
- Using the % string operator in combination with the print command, we can format our output text.

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

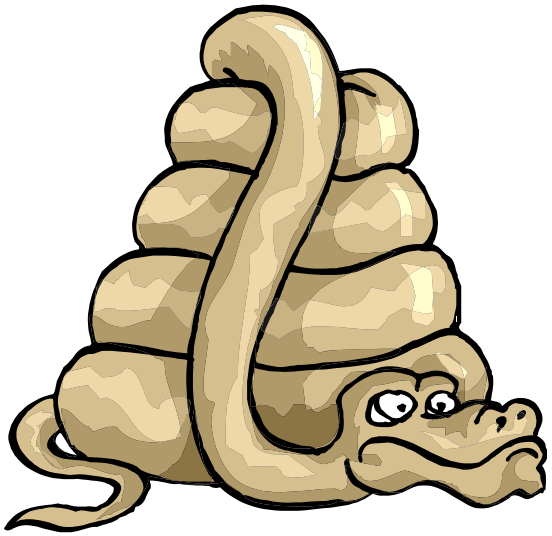
"Print" automatically adds a newline to the end of the string. If you include a list of strings, it will concatenate them with a space between them.

```
>>> print "abc"
abc
```

```
>>> print "abc", "def"
abc def
```

- Useful trick: `>>> print "abc",` doesn't add newline just a single space

String Conversions



String to List to String

- **Join turns a list of strings into one string.**

  <separator_string>.join(<some_list>)

```
>>> ";".join( ["abc", "def", "ghi"] )
 "abc;def;ghi"
```

- **Split turns one string into a list of strings.**

  <some_string>.split(<separator_string>)

```
>>> "abc;def;ghi".split( ";" )
 ["abc", "def", "ghi"]
```

- **Note the inversion in the syntax**

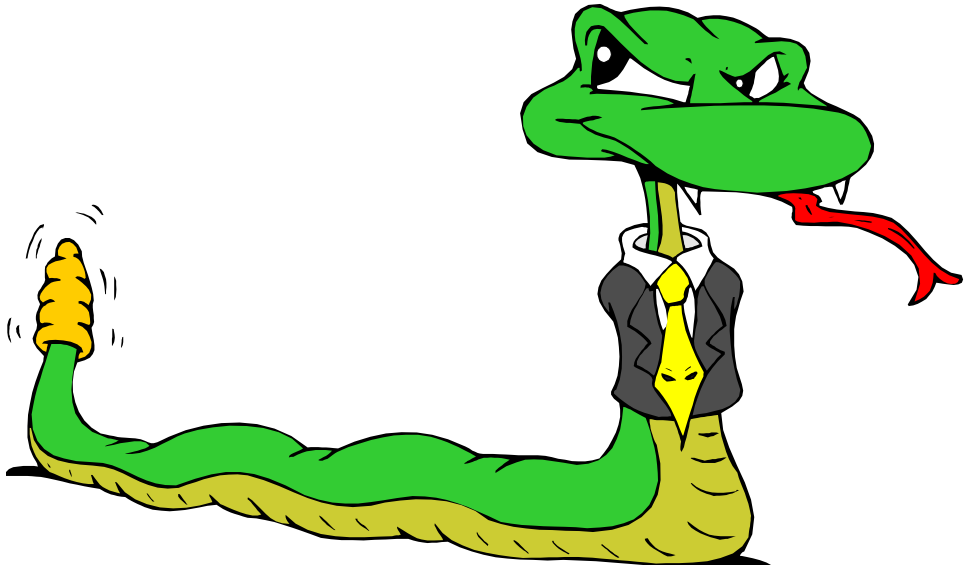
Note: Non-standard colors on this slide to help clarify the string syntax.

Convert Anything to a String

- **The built-in str() function can convert an instance of <u>any</u> data type into a string.**
  - You can define how this function behaves for user-created data types. You can also redefine the behavior of this function for many types.

```
>>> “Hello ” + str(2)
“Hello 2”
```

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

from somefile import *

from somefile import className
```

What's the difference?

<u>What</u> gets imported from the file and <u>what name</u> refers to it after it has been imported.

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

*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!**

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

from … import …

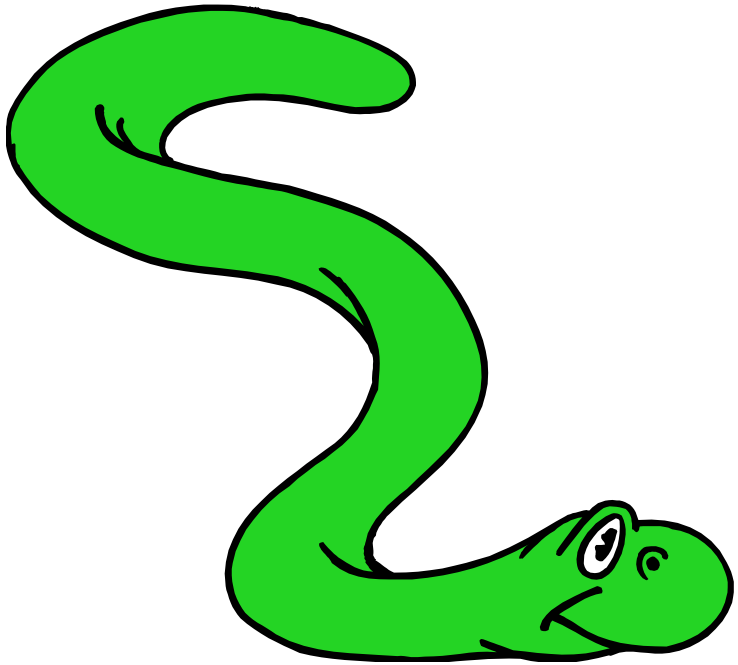
```
from somefile import className
```

- **Only the item *className* in somefile.py gets imported.**
- **After importing *className*, 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!**

```
className.method("abc")
```
← This got imported by this command.

```
myFunction(34)
```
← This one didn't.

Object Oriented Programming in Python: Defining Classes



It's all objects…

- **Everything in Python is really an object.**
  - We've seen hints of this already…
    ```
    "hello".upper()
    list3.append('a')
    dict2.keys()
    ```
  - These look like Java or C++ method calls.
  - New object classes can easily be defined in addition to these built-in data-types.
- **In fact, programming in Python is typically done in an object oriented fashion.**

Defining a Class

- **A *class* is a special data type which defines how to build a certain kind of object.**
  - The *class* also stores some data items that are shared by all the instances of this class.
  - *Instances* are objects that are created which follow the definition given inside of the class.
- **Python doesn't use separate class interface definitions as in some languages. You just define the class and then use it.**

Methods in Classes

- **Define a *method* in a *class* by including function definitions within the scope of the class block.**
  - There must be a special first argument ***`self`*** in *all* method definitions which gets bound to the calling instance
  - There is usually a special method called ***`__init__`*** in most classes
  - We'll talk about both later…

A simple class definition: *student*

```
class student:
  """A class representing a student."""
  def __init__(self,n,a):
      self.full_name = n
      self.age = a
  def get_age(self):
      return self.age
```

Creating and Deleting Instances

Instantiating Objects

- **There is no "new" keyword as in Java.**
- **Merely use the class name with () notation and assign the result to a variable.**
- **`__init__` serves as a constructor for the class. Usually does some initialization work.**
- **The arguments passed to the class name are given to its `__init__()` method.**
  - So, the __init__ method for student is passed "Bob" and 21 here and the new class instance is bound to b:

```
b = student("Bob", 21)
```

Constructor: __init__

- **An `__init__` method can take any number of arguments.**
  - Like other functions or methods, the arguments can be defined with default values, making them optional to the caller.

- **However, the first argument `self` in the definition of __init__ is special…**

Self

- **The first argument of every method is a reference to the current instance of the class.**
  - By convention, we name this argument ***`self`***.

- **In `__init__`, *`self`* refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called.**
  - Similar to the keyword *this* in Java or C++.
  - But Python uses *self* more often than Java uses *this*.

Self

- Although you must specify ***self*** explicitly when ***defining*** the method, you don't include it when ***calling*** the method.
- Python passes it for you automatically.

Defining a method:

(this code inside a class definition.)

```
def set_age(self, num):
    self.age = num
```

Calling a method:

```
>>> x.set_age(23)
```

Deleting instances: No Need to "free"

- **When you are done with an object, you don't have to delete or free it explicitly.**
  - Python has automatic garbage collection.
  - Python will automatically detect when all of the references to a piece of memory have gone out of scope. Automatically frees that memory.
  - Generally works well, few memory leaks.
  - There's also no "destructor" method for classes.

Access to Attributes and Methods

Definition of student

```
class student:
  """A class representing a student."""
  def __init__(self,n,a):
      self.full_name = n
      self.age = a
  def get_age(self):
      return self.age
```

Traditional Syntax for Access

```
>>> f = student (“Bob Smith”, 23)

>>> f.full_name     # Access an attribute.
“Bob Smith”

>>> f.get_age()      # Access a method.
23
```

File Processing with Python

This is a good way to play with the error handing capabilities of Python. Try accessing files without permissions or with non-existent names, etc.

You'll get plenty of errors to look at and play with!

```
fileptr = open('filename')
somestring = fileptr.read()
for line in fileptr:
   print line
fileptr.close()
```