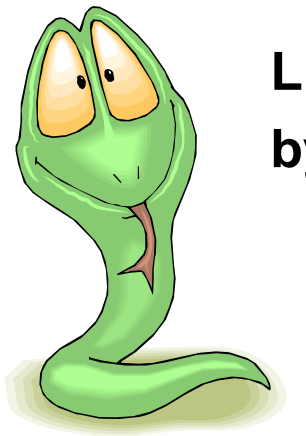
Python: A Simple Tutorial

Adapted from slides for CIS 391 by Matt Huenerfauth
University of Pennsylvania

Lightly edited further for UVic SENG 265
by Nigel Horspool & Michael Zastre (University of Victoria)



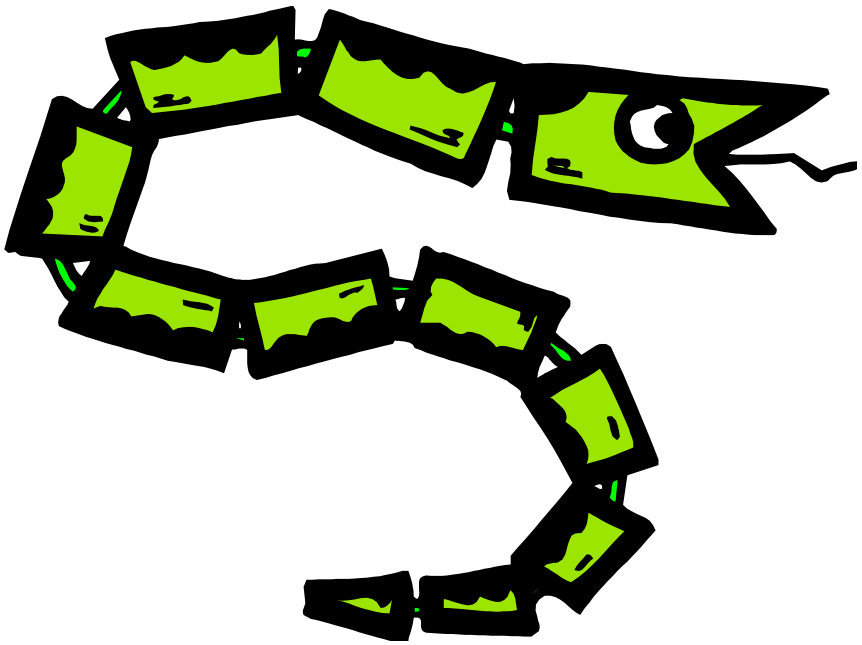
Python

- Python is an open source scripting language.
- Developed by Guido van Rossum in the early 1990s
- Named after Monty Python
- Available for download from http://www.python.org

Why Python?

- Supports Object-Oriented style of programming…
- … but you don't always need to use it
- Much less verbose than Java
- Cleaner syntax than Perl
- Built-in datatypes for strings, lists, and more
- Strong numeric processing capabilities: matrix operations, etc.
- Suitable for probability and machine learning code
- Powerful regular expressions library

The Basics



A Code Sample

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

Understanding the Code…

- **Assignment uses `=` and comparison uses `==`.**
- **For numbers `+ - * / %` behave 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 as in C or Java (i.e., do not use &&, ||, !)**
- **The basic printing function 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.
- **Block structure is denoted by indentation.**

Basic Datatypes

- **Integers (default for numbers)**

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

- **Floats**

```
x = 3.456
```

- **Strings**
  - Can use double- or single-quotes to delimit strings.
    `"abc"  'abc'` (Same thing.)
  - Unmatched quotation marks can occur within the string.
    `"matt's"`
  - Use triple double-quotes for multi-line strings or strings than contain both ′ and ″ inside them:
    `"""a'b"c"""`

Whitespace

White space 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 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.)**
- **Tip: Configure your editor to use spaces for indents (i.e., not tabs!)**

Comments

- Start comments with # – the rest of line is ignored.
- (This is a bit like "//" in Java and C++)
- 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 make use of such documentation strings, therefore it is 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, Strings, and Lists

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 Tuple (part 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 Tuple (part 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:

```
>>> tuple2 = tuple1          # 2 names refer to 1 ref
                             # Changing one affects both

>>> tuple2 = tuple1[:]       # Two independent copies,
                             # two refs
```

Note that [:] uses square brackets, but our tuple here uses parentheses.

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**

```
>>> s = 'abcde'
>>> 'c' in s
True
>>> 'cd' in s
True
>>> 'ac' in s
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'

>>> "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'
```