

Python Programming

Week Two, Class One



Reading

- Python Programming by John Zelle
 - For beginners
 - First chapters from Google Books
- Dive into Python, by Mark Pilgrim
 - For more experienced programmers
 - Free on-line

Outline

- The interpreter
- Defining functions
- Numbers
- Strings
- Programming cycle

Python interpreter

- Runs python programs
 - Interprets and evaluates statements
- Manages memory, freeing you to think about higher-level problems
- Python is designed to be user friendly.
- When you launch and use the interpreter, you create an evaluation environment – a Python session
- Use the `dir` command to find out what variables and objects exist in the current environment.

Programming

- In programming we deal with two kinds of entities:
 - Procedures – functions, directions for doing computational work
 - Data – “stuff” we do the work on
 - In python, all functions and data are objects (as with most object-oriented languages)
- An Example:
 - `>>> # a literal expression, the number 5`
 - `>>> open # a procedure`

Some definitions : Evaluation environment

- Launch python in the terminal.
 - This creates a new **evaluation environment**
 - Type something, press return
 - Python evaluates what you typed, in the context of the current **environment**

```
>>> 5 + 4
9
>>> a = 5
>>> a
5
```

This is an expression that python evaluates.

This is another expression (an **assignment**) that assigns a value (5) to the name “a” “a” is a **variable** and 5 is its **value**. From now on, the interpreter evaluates “a” as the value 5. (Unless you assign it a new value.)

Terms

- **Procedures** operate on **data**.
- A procedure accepts **operands**.
 - Operands are also called **arguments**, **parameters**

```
>>> 5 + 6
11
>>> fn = 'gene_info.txt'
>>> fh = open(fn)
```

`+` is an **operator** the **literal** values 5 and 6 are **operands**

`=` is an **operator** (an **assignment operator**) `'gene_info.txt'` is its **operand**

`open` is an **operator** and `fn` is its **operand** `fn` **evaluates** to `gene_info.txt`

The interpreter first substitutes the value of the variable `fn` before passing it to `open`

In python, the `def` operator defines new procedures.

- The `def` operator names a procedure, which you can then invoke (like `open`)

```
>>> def sayhi():
...     print "Hi!"
...
>>> dir()
['__builtins__', '__doc__', '__name__', 'sayhi']
>>> sayhi
<function sayhi at 0x68d30>
>>> sayhi()
Hi!
```

Tab: indicates the expression belongs in the block of expressions belonging to the procedure named `sayhi`

Type return to close the block

list entities existing in the evaluation environment

evaluates to the function defined as `sayhi`, stored at memory address `0x68d30` (hexadecimal)

invokes the procedure

Operands to user-defined procedures

```
>>> def sayhi(name):
...     print "Hi!" + name
...
>>> sayhi("Ann")
Hi!Ann
>>> dir()
['_builtins_', '__doc__', '__name__', 'sayhi']
>>> del(sayhi)
>>> sayhi("Ann")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sayhi' is not defined
>>> []
```

Annotations:

- redefine sayhi (points to `def sayhi(name):`)
- now it accepts an operand (points to `name`)
- name is a **bound variable** within the **scope** of this procedure (points to `name`)
- now, we can pass a operand (literal value "Ann" - a string) to sayhi (points to `sayhi("Ann")`)
- we can also delete it from the environment (points to `del(sayhi)`)

Functions can return values

```
>>> def subtract_five(n):
...     value = n - 5
...     return value
...
>>> subtract_five(10)
5
>>> b = subtract_five(10)
>>> b
5
```

Annotations:

- the return value (points to `5` from `subtract_five(10)`)
- assignment (points to `b =`)

Functions can also have side effects

- For example, `sayhi` just printed text to the terminal.
 - It didn't return a value.
 - We can re-define it to do both
- ```
>>> def sayhi():
... print "hello"
... return "howareyou?"
...
>>> sayhi()
hello
'howareyou?'
>>> a = sayhi()
hello
>>> a
'howareyou?'
>>> []
```
- Annotations:
- returned value (points to `'howareyou?'`)
  - the side effect (points to `hello`)

## More about scope

- Scope: refers to the environment in which a variable has a given value.
- A variable defined in a function is a local variable & has a scope local to that function
- What happens if you use a name already defined in the environment?

## Local variable (a) masks variable defined in the containing evaluation environment

```
>>> a = 50
>>> def addup(b):
... a = 5
... return a + b
...
>>> addup(5)
10
>>> a
50
```

## Core types

- Numbers
  - Integers
  - Floats
- Strings (sequences)
- Dictionaries (hashtables)
- Lists (sequences)
- Tuples (sequences)
- There are more, but these are the ones you will work with most.

## Numbers

- Floating-point number
  - A number with a fractional part
    - 1.003
- Integer
  - A number without a fractional part
- How do computers represent numbers?

## How python handles integers

- There are infinitely-many integers, but only a finite range of python ints.
- Why? Computers store data using electronic switches, which are either on or off
- Sequences of bits to hold data, values 0 and 1

| Bit 2 | Bit 1 |
|-------|-------|
| 0     | 0     |
| 0     | 1     |
| 1     | 0     |
| 1     | 1     |

Two bits can store 4 values  
Three bits can store ???  
32 bits can store ???

## What's the biggest int?

- Typical PCs use 32 or 64 bits to represent ints.
- Launch python. Multiple 2 times itself 32 times.
- Write a function – power

```
def power(base,exponent):
 result = 1
 for i in range(1,exponent):
 result = base * result
 return result
```

Note! This function yields the wrong answer. Tell me why..  
**Hint:** what does range return?

## Finding the biggest int

```
>>> import week2class1 as w
>>> dir(w)
['__builtins__', '__doc__', '__file__', '__name__', 'power']
>>> w.power(2,32)
4294967296L
>>> w.power(2,31)
2147483648L
>>> w.power(2,30)
1073741824
```

These are Long ints

It's between  $2^{30}$  and  $2^{31}$ .

Homework Problem: Calculate the biggest int and prove your answer is correct.

## Floats

- Represent numbers with fraction parts.
- Python sometimes uses scientific notation to display floats:

```
>>> 8/3.*10**100
2.6666666666666666e+100
```

significand → ← exponent

- Significand – “significant digits”
- Notation e+100 means  $10^{100}$
- Computer stores exponent, significand as ints

## Floats

- Precision and size are limited.

```
>>> pow(2,100)
1267650600228229401496703205376L
>>> pow(2.,100)
1.2676506002282294e+30
>>> pow(2.,1000)
1.0715086071862673e+301
>>> pow(2.,2000)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
OverflowError: (34, 'Result too large')
```

# Strings

- Strings are sequences of characters
  - There are several ways to create a string

```
>>> 'a string'
'a string'
>>> "a string"
'a string'
>>> """ a string
...
... across multiple lines
...
... """
'a string\n\n\nacross multiple lines\n\n'
>>> □
```

Use triple quotes to define a string containing new-line characters

# Strings are computational entities that can do work:

- They have methods attached to them!
- Technically speaking, strings are objects that have methods:

```
>>> s = "Hi.\n\nHow are you?"
>>> s.split()
['Hi.', 'How', 'are', 'you?']
>>> s.upper()
'HI.\n\nHOW ARE YOU?'
>>> s
'Hi.\n\nHow are you?'
```

First, define a string literal. Note escaped newline character.

Note: calling upper on a string returns new string...it doesn't change the original string

# What other methods do strings have?

```
>>> s
'Hi.\n\nHow are you?'
>>> dir(s)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__ge__', '__getattr__', '__getitem__', '__getnewargs__', '__getslice__',
'__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmul__', '__rmod__', '__setattr__', '__str__', 'capitalize', 'center',
'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'index', 'isalnum',
'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Some of these are data.

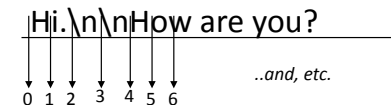
Some of these are methods.

Don't access data or methods with names that start with `__` (double underscore). By convention, these are private!! They could change in later versions of python.

# Strings are also sequences

```
>>> s
'Hi.\n\nHow are you?'
>>> s[0:1]
'H'
>>> s[0:2]
'Hi'
>>> s[:-1]
'Hi.\n\nHow are you'
>>> s[:-3]
'Hi.\n\nHow are y'
>>> s[2:]
'.\n\nHow are you?'
>>> □
```

- Access parts of strings using **slicing**
- Numbering starts at 0:



## Strings are immutable

- Once created, you can't change them.
- They differ from lists, another type of sequence.

```
>>> s
'Hi.\n\nHow are you?'
>>> s[1]='a'
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

## Actually, everything in python is an object...even numbers

- But note: no non-private methods or data:

```
>>> f = 1234.5678 ← Defining a new float.
>>> f
1234.5678 A float literal.
>>> dir(f)
['__abs__', '__add__', '__class__', '__coerce__', '__delattr__', '__div__',
['__divmod__', '__doc__', '__eq__', '__float__', '__floordiv__', '__ge__',
['__getattr__', '__getformat__', '__getnewargs__', '__gt__', '__hash__',
['__init__', '__int__', '__le__', '__long__', '__lt__', '__mod__', '__mul__',
['__ne__', '__neg__', '__new__', '__nonzero__', '__pos__', '__pow__', '__radd__',
['__rdiv__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__',
['__rfloordiv__', '__rmod__', '__rmul__', '__rpow__', '__rsub__', '__rtruediv__',
['__setattr__', '__setformat__', '__str__', '__sub__', '__truediv__']
>>> f.__str__
<method-wrapper '__str__' of float object at 0x835dc4>
>>> f.__str__()
'1234.5678'
```

## Define & edit functions in files

- Use `import` to load functions from files.
- Files containing functions: **module** files.
- Typical process:
  - Write a function in a file (using emacs)
    - `my_funcs.py`
  - Import it into your environment with `import`
    - `>>> import my_funcs`
  - Invoke it using “live” data you define in the environment
    - `>>> my_funcs.do_something_useful(data)` ← Invoking a function
  - Edit the function, if necessary
  - Re-import the function using `reload`

## How python interpreter finds files containing your code

- `>>> import mycode`
  - Interpreter looks for a file called `mycode.py` in the current working directory
  - Or...in directories specified by your `PYTHONPATH` shell variable
- If you modify `mycode.py`, use `reload` to refresh the interpreter's environment
  - `>>> reload(mycode)`

## Setting up the interpreter

- Open a Terminal
- Set up your PYTHONPATH

When I work on a shared computer, I like to set up a directory just for me, called "loraine\_src." I point my PYTHONPATH to directories in that location:

```
export SRC=$HOME/loraine_src
export PYTHONPATH=SRC:.
```

When I'm done, I copy my files onto a USB drive, or check them into a version control system, and *delete*.

## Back to programming

- Your process of programming should involve writing functions, testing these in the interpreter
- Especially good practice as you learn the syntax
- You can test constructs as you go...

## How to get help

- For help, visit: [www.python.org/doc](http://www.python.org/doc)
- Within the interpreter, you can also get help on a module or a command:

```
>>> help(urllib)
>>> help(urllib.urlopen)
```