



# CS 11 Python track: lecture 1

---

- Preliminaries

- Need a CS cluster account
  - [http://www.cs.caltech.edu/cgi-bin/sysadmin/account\\_request.cgi](http://www.cs.caltech.edu/cgi-bin/sysadmin/account_request.cgi)
- Need to know UNIX
  - ITS tutorial linked from track home page
- Recommended: CS 1
- Track home page:
  - [www.cs.caltech.edu/courses/cs11/material/python](http://www.cs.caltech.edu/courses/cs11/material/python)



# Assignments

---

- 1st assignment is posted now
- Due one week after class, midnight
- Late penalty: 1 mark/day
- Redos



# Redos

---

- 1st redo = 1 mark off
- 2nd redo = 1 to 2 more marks off
- 3rd redo = 1 to 3 more marks off
- No 4th redo! Grade - 6 mark penalty



# Passing

---

- Need average of 7/10 on labs
- 7 labs → 49/70 marks



# Other administrative stuff

---

- See admin web page:

<http://www.cs.caltech.edu/courses/cs11/material/python/admin.html>

- Covers how to submit labs, collaboration policy, grading, etc.



# Textbook

---

- None required
- "Learning Python" by Mark Lutz
- Most people learn from online docs
  - links on web site



# Why learn Python?

---

- "Scripting language"
- Very easy to learn
- Interactive front-end for C/C++ code
- Object-oriented
- Powerful, scalable
- Lots of libraries
- Fun to use



# Python syntax

---

- Much of it is similar to C syntax
- Exceptions:
  - missing operators: `++`, `--`
  - no `{ }` for blocks; uses whitespace
  - different keywords
  - lots of extra features
  - no type declarations!



# Starting and exiting Python

---

```
% python
```

```
Python 2.4.1 ...
```

```
>>> print "hello"
```

```
hello
```

```
>>> ^D
```

```
%
```



# Simple data types

---

- Numbers
  - integer
  - floating-point
  - complex!
- Strings
  - characters are strings of length 1
- Booleans are **0/1** (or **False/True**)



# Simple data types: operators

---

- `+` `-` `*` `/` `%` (like C)
- `+=` `-=` etc. (no `++` or `--`)
- Assignment using `=`
  - but semantics are different!  
`a = 1`  
`a = "foo" # OK`
- Can also use `+` to concatenate strings



# Compound data types (1)

---

- Lists:

```
a = [1, 2, 3, 4, 5]
```

```
print a[1] # 2
```

```
some_list = []
```

```
some_list.append("foo")
```

```
some_list.append(12)
```

```
print len(some_list) # 2
```



## Compound data types (2)

---

- Dictionaries:

- like an array indexed by a string

```
d = { "foo" : 1, "bar" : 2 }
```

```
print d["bar"] # 2
```

```
some_dict = {}
```

```
some_dict["foo"] = "yow!"
```

```
print some_dict.keys() # ["foo"]
```



# Compound data types (3)

---

- Tuples:

```
a = (1, 2, 3, 4, 5)
```

```
print a[1] # 2
```

```
empty_tuple = ()
```

- Difference between lists and tuples:
  - lists are mutable; tuples are immutable
  - lists can expand, tuples can't
  - tuples are slightly faster



# Compound data types (4)

---

- Objects:

```
class Thingy:
```

```
    # next week's lecture
```

```
t = Thingy()
```

```
t.method()
```

```
print t.field
```

- Built-in data structures (lists, dictionaries) are also objects
  - though internal representation is different



# Control flow (1)

---

- **if, if/else, if/elif/else**

```
if a == 0:
```

```
    print "zero!"
```

```
elif a < 0:
```

```
    print "negative!"
```

```
else:
```

```
    print "positive!"
```



## Control flow (2)

---

- Notes:
  - blocks delimited by indentation!
  - colon (:) used at end of lines containing control flow keywords



## Control flow (3)

---

- **while** loops

```
a = 10
```

```
while a > 0:
```

```
    print a
```

```
    a -= 1
```



## Control flow (4)

---

- **for** loops

```
for a in range(10):  
    print a
```

- really a "foreach" loop



## Control flow (5)

---

- Common **for** loop idiom:

```
a = [3, 1, 4, 1, 5, 9]
for i in range(len(a)):
    print a[i]
```



## Control flow (6)

---

- Common **while** loop idiom:

```
f = open(filename, "r")
```

```
while True:
```

```
    line = f.readline()
```

```
    if not line:
```

```
        break
```

```
    # do something with line
```



## Aside: `open()` and `file()`

---

- These are identical:

```
f = open(filename, "r")  
f = file(filename, "r")
```
- The `open()` version is older
- The `file()` version is the recommended way to open a file now
  - uses object constructor syntax (next lecture)



## Aside 2: file iteration

---

- Instead of using **while** loop to iterate through file, can write:

```
f = file("some_file", "r")
```

```
for line in f:
```

```
    # do something with line...
```

- More concise, generally considered better



# Control flow (7): odds & ends

---

- **continue** statement like in C

```
a = 0
```

```
while a < 10:
```

```
    a += 1
```

```
    if a % 2 == 0:
```

```
        continue # to next iteration
```

```
    else:
```

```
        print a
```



# Control flow (7): odds & ends

---

- **pass** keyword:

```
if a == 0:
```

```
    pass    # do nothing
```

```
else:
```

```
    # whatever
```



# Defining functions

---

```
def foo(x):  
    y = 10 * x + 2  
    return y
```

- All variables are local unless specified as **global**
- Arguments passed by value



# Executing functions

---

```
def foo(x):
```

```
    y = 10 * x + 2
```

```
    return y
```

```
print foo(10)    # 102
```



# Comments

---

- Start with # and go to end of line
- What about C, C++ style comments?
  - NOT supported!



# Writing standalone scripts

---

- Can execute any file like this:

```
% python myprog.py
```

- Might want file to be directly executable, so...
- at top of file, write this:

```
#! /usr/bin/env python
```

```
# code goes here...
```

- Then make file executable:

```
% chmod +x myprog.py
```

```
% myprog.py
```



# File naming conventions

---

- python files usually end in `.py`
- but executable files usually don't have the `.py` extension
- modules (later) should always have the `.py` extension



# Take a deep breath...

---

- Almost done! ;-)
- More on strings
- Modules
- Command-line arguments
- File I/O



# Strings and formatting

---

```
i = 10
```

```
d = 3.1415926
```

```
s = "I am a string!"
```

```
print "%d\t%f\t%s" % (i, d, s)
```

```
print "no newline",
```



# Modules (1)

---

- Access other code by importing modules

```
import math
```

```
print math.sqrt(2.0)
```

- Or:

```
from math import sqrt
```

```
print sqrt(2.0)
```



## Modules (2)

---

- or:

```
from math import *  
print sqrt(2.0)
```

- Can import multiple modules on one line:

```
import sys, string, math
```

- Only one "from x import y" per line



# Modules (3)

---

- NOTE!

```
from some_module import *
```

- should be avoided
- dumps all names from `some_module` into local namespace
- easy to get inadvertent name conflicts this way



# Modules (4)

---

- Code you write in file `foo.py` is part of module `"foo"`

- Can import this code from within other files:

```
import foo
```

```
# code that uses stuff from foo
```



# Command-line arguments

---

```
import sys

print len(sys.argv) # NOT argc

# Print all arguments:

print sys.argv

# Print all arguments but the program
# or module name:

print sys.argv[1:] # "array slice"
```



# File I/O

---

```
f = file("foo", "r")  
line = f.readline()  
print line,  
f.close()  
  
# Can use sys.stdin as input;  
# Can use sys.stdout as output.
```



# Whew!

---

- First assignment is easy
- Next week: classes and objects