02-0: **Python**

- Name "python" comes from Monte Python's Flying Circus
  - Most python references use examples involving spam, parrots (deceased), silly walks, and the like
- Interpreted language
- Type in an expression, returns the value
- Use Python like a calculator
- Variables don't need to be declared, type is inferred by assigning a value

02-1: **Why Python is Cool**

- Easy to use & read
- Strongly typed, with inferred types
- First order programming
  - Everything is an object
  - Functions as data
- Lots of powerful built-in libraries
  - File processing (including URLs)
  - regular expressions
  - GUIs

02-2: **Python as Calculator**

- All the standard operators
  - +, -, *, /, %, ** or pow for x^y
- Assigning a value to a variable declares it
  - Type is inferred from value assigned
- Coercion, just like
  - 3 + 4.0 / 2
  - 3 / 2 = ?

02-3: **Datatypes: Numbers**

- Integers (longs in C) 1, -32, 5612
- Long integers (unlimited size) 333422395954556L
- floats (doubles in C) 1.23 3.1e+15
- Octal and Hexadecimal 0143, 0x3aff3
- Complex numbers (3.0 + 5j)

02-4: **Datatypes: Strings**

- Denoted with ” or ”” (equivalent)

```
>>> "spam"
'spam'
>>> 'spam'
'spam'
```

- Can mix and match, helpful when want ’ or ” in a string:

```
>>> "The parrot was 'dead'"
"The parrot was 'dead'"
>>> 'The parrot was "dead"'
'The parrot was "dead"'
```

02-5: **Datatypes: Strings**

- Multi-line strings using ”””

```
>>> """This is a
multiline string"""
'This is a\nmultiline string'
```

- Handy for function comments (more on this in a bit)

02-6: **Datatypes: Strings**

- Access individual elements using subscripts:

```
>>> x = "Hello There"
>>> x[3]
'l'
```

(Note that ’l’ is not a character, it is a string of length 1 (no chars in python))

- Also use slices:

```
>>> x = "Hello There"
>>> x[3:5]
'llo'
```

02-7: **Datatypes: Strings**

- Negative indicies in slices count from the end of the string:

```
>>> x = "Hello There"
>>> x[0:-3]
'Hello The'
```

- Think of the indices as pointing between charaters:

```
 +---+---+---+---+---+
 | S | p | a | m | ! |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

02-8: **Datatypes: Strings**


```
 +---+---+---+---+---+
 | S | p | a | m | ! |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```


- What should this return?

```
>>> x = "Hello There"
>>> x[-1:-5]
```

02-9: **Datatypes: Strings**

- Can concatinate strings using "+" (just like java)

```
>>> x = "cat"
>>> y = "dog"
>>> x + y
'catdog'
```

- Repitition using *

```
>>> "cat" * 3
'catcatcat'
```

02-10: **Datatypes: Strings**

- Strings are immutable

```
>>> x = "cat"
>>> x[1] = "o"
ERROR
```

- How could we change the elemet at index 1 to an "o"?

02-11: **Datatypes: Strings**

- Strings are immutable


```
>>> x = "cat"
>>> x[1] = "o"
ERROR
```


- How could we change the elemet at index 1 to an "o"?

```
>>> x = "cat"
>>> x = x[0:1] + 'o' + x[2:3]
>>> x
'cot'
```

- Note that this is a bit wasteful, creates lots of strings (more on how to do string manipulation efficiently in a bit ...)

02-12: **Datatypes: Lists**

- Items between [and], separated by commas are lists
- Lists are heterogeneous

```
>>> [1, 2, 3, 4]
[1, 2, 3, 4]
>>> [3, "a",  4.5, 3+4j]
[3, 'a', 4.5, (3+4j)]
```

02-13: **Datatypes: Lists**

- Access elements with [], but lists are mutable (unlike strings)

```
>>> x = [1,  2, 3, 4]
>>> x[2]
3
>>> x[2] = 99
>>> x
[1, 2, 99, 4]
```

02-14: **Datatypes: Lists**

- Python makes list processing *very* easy

```
>>> x = [1, 2, 3]
>>> x.append("car")
>>> x
[1, 2, 3, 'car']
>>> x[2] = [1,2,3,4]
>>> x
[1, 2, [1, 2, 3, 4], 'car']
```

02-15: **Datatypes: Lists**

- append(), pop() – stacks and queues
- +, *, append, extend, sort, reverse
- Use slices (just like strings)

```
>>> x = [1,2,3,4]
>>> x[1:2] = [5,6,7,8]
>>> x
[1, 5, 6, 7, 8, 3, 4]
```

02-16: **Datatypes: Lists**

- List variables store reference:

```
>>> x = [1,2,3,4]
>>> y = x
>>> y[1] = 99
>>> x
[1, 99, 3, 4]
```

- Get a copy by using a slice

```
>>> x = [1,2,3,4]
>>> y = x[:]
>>> y[1] = 99
>>> x
[1, 99, 3, 4]
```

02-17: **== vs. is**

- Python does a good job of doing "what you want"
- "==" is value-equality, not reference equality
- "is" is reference equality

```
>>> x = [1,2,3,4]
>>> y = [1,2,3,4]
>>> z = x
>>> x == y
True
>>> x is y
False
>>> x is z
True
```

02-18: **Tuples**

- Immuable lists
- use ( ) instead of [ ]
  - () empty tuple
  - (3,2) tuple with two elements
- What about singletons?
  - (3) is just 3 with parens
  - (3,) is a singleton tuple
- Otherwise, just like lists

02-19: **Tuples**

- Can use tuples for multiple assignment
- Handy for swapping (also for returning > 1 value)

```
>>> spam, chips = 3,4
>>> spam, chips = chips, spam
>>> spam
4
>>> chips
3
```

02-20: **Datatypes: Dictionaries**

- Like hash tables
- Denoted with { }
- Accessed like arrays

```
>>> x = { }
>>> x["cat"] = 3
>>> x["dog"] = "mouse"
>>> x[4] = 'pipsqueak'
```

02-21: **Datatypes: Dictionaries**

- Can create a dictionary on a single line:

```
>>> x = { "green" : "eggs", 3 : "blind mice"}
>>> x["green"]
'eggs'
>>> x["newentry"] = "new value"
```

02-22: **Datatypes: Dictionaries**

- Can have nested dictionaries

```
>>> x = { "red" : 3, "complex" : { "blue" : 4 } }
>>> x["red"]
3
>>> x["complex"]
{ "blue" : 4 }
>>> x["complex]["blue"]
4
```

02-23: **Datatypes: Dictionaries**

- "keys" method returns a list of keys in a dictionary
- Add elements to a dictionay by assignment
- Delete keys using del

```
>>> x = { "red" : 3, "blue" : 4 }
>>> x["green"] = 5
>>> x
{ 'red' : 3, 'blue' : 4, 'green' : 5 }
>>> del x['blue']
>>> x
{ 'red' : 3, 'green' : 5 }
```

02-24: **Multiple Lines**

- No separators (semicolons, etc)
- No begin/end, {, } to define blocks
- One statement per line, blocks defined by indentation

02-25: **Control Structures: if**

```
if <test>:
   <statement>
   <statement>
elif:
   <statement>
   <statement>
elif:
   <statement>
   <statement>
else:
   <statement>
   <statement>
```

02-26: **Control Structures: while**

```
while <test>:
   <statement>
   <statement>
   <statement>
```

- break, continue
  - just like java/C/C++

02-27: **Booleans in Python**

- False:
  - False (built in, careful of case!)
  - 0, 0.0 (be careful of rounding errors!)
  - () (empty tuple)
  - `[]` (empty list)
  - {} (empty dictionary)
  - "" (empty string

- True:
  - Anything else

02-28: **Booleans in Python**

- a and b
  - if a is true, return b, else return a
- a or b
  - if a is true, return a, else return b

02-29: **and-or trick**

- Can get C-style (test ? x : y)
  - test and x or y
- Examples ...
- When does this break?

02-30: **and-or trick**

- Fixing the and-or trick:
  - (test and [x] or [y])[0]
- What does this do?
- Do we have the same problem?

02-31: **Iterators**

- for loop:

```
>>> lst = [1, 2, 3, 4]
>>> for x in lst:
   print x,
1 2 3
```

  - Trailing , supresses end-of-line
  - For loop only iterates over a data structure
  - Use "range([low],high,[skip])" to iterate over a range

02-32: **Iterators**

- Dictionaries:

```
>>> d = {'a': 1, 'b':2, 'c': 3 }
>>> for key in d:
   print key,
a b c
>>> for key, value in d.iteritems():
   print key, value
a 1
b 2
c 3
```

02-33: **Membership**

- test with `in <data structure>`

```
>>> x = [1, 2, 3, 4]
>>> 2 in x
True
>>> 5 in x
False
>>> y = {"car": 1, "dog" : 2}
>>> "car" in y
True
>>> 1 in y
False
```

02-34: **Functions**

```
def <name>(params):
    <body>
```

- Params are all pass-by-value (like C/Java)
- Return statements work just like C/Java
- Can use tuples to return > 1 value from a function

02-35: **Functions**

```
def fib(n):
  if n <= 2:
    return 1
  else:
    return fib(n-1) + fib(n-2)

def fib2(n):
  if n <= 2:
    return (1,1)
  else:
    (prev, prevPrev) = fib2(n-1)
    return prev+prevPrev, prev
```

02-36: **Function comments**

```
def <name>(params):
"""Comment that describes
the function """
    <body>
```

- Comment is part of the function itself
- Can be accessed with help(functionname)

02-37: **Function parameters**

- Functions can have optional praramters

- Can call functions using name of the parameter
- Can have variable numbers of parameters
  - *args, **args

02-38: **Modules**

- Each .py file is a "module"
- Can load "module.py" with "import module"
- Module needs to be in a location described by PYTHONPATH enviornment variable
  - PYHONPATH has same syntax as standard PATH
  - Path stored in sys.path, can modify at runtime
- Need to use "module" when calling functions
  - from <module> import <symbol>
  - from <module> import *

02-39: **Python scripts**

- When you import a module, execute the entire file
  - def's generate functions
  - have any code at all – executed when module is run
- .py files can be scripts (to be run from the commmand line), or modules (imported by other python programs). We can have the same .py file serve 2 purposes
  - The symbol `__name__` will have the value `__main__` if and only if file is being used as a script

```
if __name__ == "__main__":
    <run main program of script>
```

02-40: **File Handling**

- outfile = file('fname', 'w'), infile = file('fname', 'r')
  - 'r' is default, can be left out
- S = infile.read() – reads entire file into string S
- S = infile.read(n) – reads first n lines into S
- S = infile.readline() – reads one line into S
- L = infile.readlines() – reads while file into a list of strings
  - Unless the file is really large, better to read all at once with read() or readlines(), and then process the strings

02-41: **URLs**

```
>>> Import urllib
>>> sock = urllib.urlopen("http://cs.usfca.edu/")
>>> htmlSource = sock.read()
>>> sock.close()
>>> print htmlSource
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>Department of Computer Science</title>
  <link rel="stylesheet" type="text/css" href="/cs.css">
  <link rel="shortcut icon" type="image/ico" href="/favicon.ico">
</head>
... etc
```

02-42: **Regular Expressions**

- Dive into Python has a good explaination
- Dive in, and come to me with questions
- Spend lecture time on regular expressions if there is classwide confusion