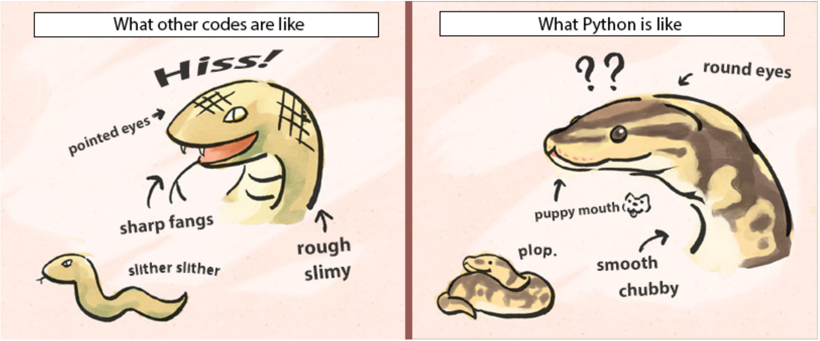
Basic Training



Outline

- Hello World!
- Calculator/basic math
- Strings
- Variables
- Basic control statements
  - Indentation!

C++

```
#include <iostream>
int main()
{
  std::cout << "Hello World!" << std::endl;
}
```

```
BootCamp> g++ -o hello hello.cpp
BootCamp> ./hello
Hello World!
BootCamp>
```

Java

```
class HelloWorld {
    static public void main( String args[] ) {
        System.out.println( "Hello World!" );
    }
}
```

```
BootCamp> javac hello.java
BootCamp> java HelloWorld
Hello World!
BootCamp>
```

Fortran

```
      PROGRAM HELLO
      WRITE (*,100)
      STOP
 100  FORMAT (' Hello World! ' /)
      END
```

```
BootCamp> g77 -o hello hello.f
BootCamp> ./hello
Hello World!
BootCamp>
```

- Examples of compiled languages

Scripted

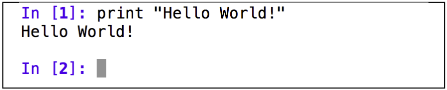
Python

```
print "Hello World!"
```

```
BootCamp> python hello.py
Hello World!
BootCamp>
```

Interactive

```
BootCamp > python
>>> print "Hello World!"
Hello World!
>>>
```



```
In [1]: print "Hello World!"
Hello World!

In [2]:
```

2 quick points

1. Python provides an interactive way to develop code and execute scripts
2. What you do interactively is the same thing as a script

Calculator

```
>>> print 2 + 2
4
>>> 2 + 2
4
>>> print 2.1 + 2
4.1
>>> 2.1 + 2 == 4.0999999999999996
True
```

- There are int & float but not doubles
- Python stores floats as their byte representation. Therefore it is limited by the same 16-bit issues as most other languages
- When doing calculations, unless you specify, Python will store the results in the smallest byte representation

Calculator

```
>>> 2 + 2
4
>>>  2 + 2
File "<stdin>", line 1
    2+2
    ^
IndentationError: unexpected indent
>>> 2 # this is a comment and is not printed
2
>>> # this is also a comment
>>>
```

1. Indentation matters!
2. When you mess up, python is gentle
3. # starts a comment until the end of the line

Calculator

```
>>> (3.0*10.0 - 25.0)/5.0
1.0
>>> print 3.085e18*1e6 # this is a Megaparsec in units of cm!
3.085e+24
>>> t = 1.0 # declare a variable t (time)
>>> accel = 9.8 # acceleration in units of m/s^2
>>> # distance travelled in time t seconds is 1/2 a*t**2
>>> dist = 0.5*accel*t*t
>>> print dist # this is the distance in meters
4.9
>>> dist1 = accel*(t**2)/2
>>> print dist1
4.9
>>> dist2 = 0.5*accel*pow(t,2)
>>> print dist2
4.9
```

- **variables** are assigned automatically
- multiplication, division, exponents work as you expect

Calculator

```
>>> 6 / 5 ; 9 / 5 # integer division returns the floor
1
1
>>> 6 % 5 # mod operator
1
>>> 1 << 2  ## shift: move the number 1 by two bits to the left
            ## that is make a new number 100 (base 2)
4
>>> 5 >> 1  ## shift: move the number 5 = 101 (base 2) one to
            ## to the right (10 = 2)
2
>>> x=2;y=3 ##assign two variables on the same line!
>>> x|y     ## bitwise OR
3
>>> x^y     ## exclusive OR (10 ^ 11 = 01)
1
>>> x & y   ## bitwise AND
2
>>> x = x ^ y ; print x
1
>>> x += 3 ; print x
4
>>> x /= 2.0
>>> print x
2.0
```

Calculator

```
>>> # from before dist1 = 4.9 and dist = 4.9
>>> dist1 == dist
True
>>> dist < 10
True
>>> dist <= 4.9
True
>>> dist < (10 + 2j)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: no ordering relation is defined for complex numbers
>>> dist < -2.0
False
>>> dist != 3.1415
True
```

More on variables and types

None, numbers, and truth

```
>>> 0 == False
True
>>> not False
True
>>> 0.0 == False
True
>>> not (10.0 - 10.0)
True
>>> not -1
False
>>> not 3.1415
False
>>> x = None # None is something special. Not true or false
>>> None == False
False
>>> None == True
False
>>> False or True
True
>>> False and True
False
```

More on variables and types

Built in types: int, bool, str, float, complex, long...

```
>>> print type(1)
<type ‘int’>
>>> x = 2 ; type(x)
<type ‘int’>
>>> type(2) == type(1)
True
>>> print type(True)
<type ‘bool’>
>>> print type(type(1))
<type ‘type’>
>>> print type(pow)
<type ‘builtin_function_or_method’>
```

we can test whether something is a certain type with isinstance()

```
>>> isinstance(1,int)
True
>>> isinstance("spam",str)
True
>>> isinstance(1.212,int)
False
```

Strings

- Strings are sequences of characters
  - They are can are indexed and can be sliced up like arrays
  - You can concatenate strings together with the + sign
- Strings are immutable (unlike C). You cannot change a string in place (this sounds worse than it is...)
- Strings can be formatted and compared

Strings

```
>>> x = "spam" ; print type(x)
<type "str">
>>> print "hello!$ \n...my sire."
hello!
...my sire.
>>> "hello! \n...my sire."
‘hello!  \n...my sire.’
>>> "wah?!" == ‘wah?!’
True
>>> print " ‘wah?!’ said the student"
‘wah?!’ said the student
>>> print "\"wah?! \" said the student"
"wah?!" said the student
```

- backslashes (\\) start special (escape) characters:
  - \\n = newline
  - \\t = tab
  - \\a = bell
- string literals are defined with double quotes or quotes. The outermost quote type cannot be used inside the string (unless it's escaped with a backslash)

Strings

```
>>> # raw strings don’t escape characters
>>> print r’This is a raw string...newlines \r\n are ignored.’
This is a raw string...newlines \r\n are ignored.
>>> # Triple quotes are real useful for multiple line strings
>>> y = ‘‘‘For score and seven minutes ago,
you folks all learned some basic mathy stuff with Python
and boy were you blown away!’’’
>>> print y
For score and seven minutes ago,
you folks all learned some basic mathy stuff with Python
and boy were you blown away!
```

- Prepending r makes that string “raw”
- Triple quotes allow you to compose longs strings
- Prepending u makes that string “unicode”

Strings

```
>>> s = "spam" ; e = "eggs"
>>> print s + e
spameggs
>>> print s + " and " + e
spam and eggs
>>> print "green " + e + " andtext\n" + s
green eggs and
  spam
>>> print s*3 + e
spamspamspameggs
>>> print "*"*50
**************************************************
>>> print "spam" is "good" ; print "spam" is "spam"
False
True
>>> "spam" < "zoo"
True
>>> "s" < "spam"
True
```

- Concatenate strings with the + sign
- Preform multiple concatenations with the * sign
- Strings can be compared

Strings

```
>>> print ‘I want’ + 3 + ‘ eggs and no ’ + s
TypeError Traceback (most recent call last)
TypeError: cannot concatenate ‘str’ and ‘int’ objects
>>> print ‘I want ’ + str(3) + ‘ eggs and no ’ + s
I want 3 eggs and no spam
>>> pi = 3.14159
>>> print ‘I want ’ + str(pi) + ‘ eggs and no ’ + s
I want 3.14159 eggs and no spam
>>> print str(True) + ":" + ‘ I want ’ + str(pi) + ‘ eggs and no ’ + s
True: I want 3.14159 eggs and no spam
```

- You can only concatenate strings.
- You can cast other variable types to str

Strings

```
>>> faren = raw_input("Enter the temperature (in Fahrenheit): ")
Enter the temperature (in Fahrenheit): 71
>>> cent = (5.0/9.0)*(faren - 32.0)
...
TypeError: unsupported operand type(s) for -: ’str’ and ’float’
>>> faren = float(faren)
>>> cent = (5.0/9.0)*(faren - 32.0) ; print cent
21.6666666667
>>> faren = float(raw_input("Enter the temperature (in Fahrenheit): "))
Enter the temperature (in Fahrenheit): 71
>>> print (5.0/9.0)*(faren - 32.0)
21.6666666667
>>> faren = float(raw_input("Enter the temperature (in Fahrenheit): "))
Enter the temperature (in Fahrenheit): meh!
...
ValueError: invalid literal for float(): meh!
```

Strings

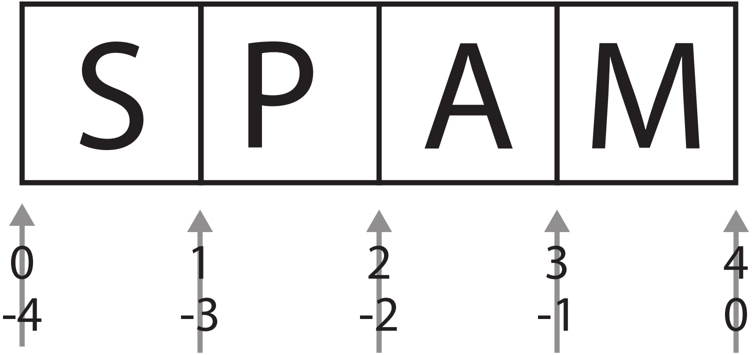
We can think of strings as arrays, unlike in C you never really need to deal with addressing characters locations in memory

```
>>> s ="spam"
>>> len(s)
4
>>> len("eggs\n")
5
>>> len("")
0
>>> s[0]
‘s’
>>> s[-1]
‘m’
```

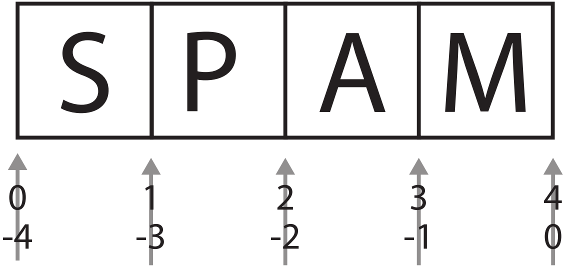
- len() give us the length of an array
- Strings are zero indexed
- Strings can also count backwards

Strings

We can think of strings as arrays, unlike in C you never really need to deal with addressing characters locations in memory



Useful for slicing: indices are between the characters



```
>>> s[0:1] # get every character between 0 and 1
‘s’
>>> s[1:4] # get every character between 1 and 4
‘pam’
>>> s[-2:-1]
‘a’
>>> ## slicing [m:n] will return abs(n-m) characters
>>> s[0:100] # if the index is beyond the len(str), you dont segfault!
‘spam’
>>> s[1:] # python runs the index to the end
‘pam’
>>> s[:2] # python runs the index to the beginning
‘sp’
```

$$s = s[:n] + s[n:] \text{ for all } n$$

Basic Control Flow

Python had pretty much all of what you need:

if...elif...else, for, while

As well as:

break, continue (within loops)

Does not have:	**Does have**:
case(explicitly) goto	pass

Flow is done within blocks (where indention matters)

```
>>> x =1
>>> if x > 0:
...   print "yo"
...else:
...   print "dude"
...
yo
```

Note colons and indentations (tabbed or spaced)

```
>>> x =1
>>> if x > 0:
...   print "yo"
...else:
...        print "dude"
...
yo
```

Indentations within the same block must be the same but not within different blocks but this is ugly and might cause confusion

One-liners

```
>>> print "yo" if x > 0 else "dude"
"dude"
```

A small program:

```
>>> x = 1
>>> while True:
...     print "yo" if x > 0 else "dude"
...     x *= -1
...
yo
dude
yo
dude
yo
dude
^Cdude
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyboardInterrupt
IndentationError: unexpected indent
```

Control-c break a loop and drops you back to the terminal

Case statements can be constructed with a bunch of if, elif,...else

```
>>> if x < 1:
...    print "t"
...  elif x > 100:
...    print "yo"
...  else:
....   print "dude"
dude
```

Ordering matters. The first block of True in an if/elif gets executed then everything else does not.

Blocks cannot be empty

```
>>> x = "fried goldfish"
>>> if x == "same for dinner":
...    print "I will destroy the universe"
... else:
...    # I'm fine with that.  I'll do nothing
...
File "<stdin>", line 5
^
IndentationError: expected an indented block
>>>
```

pass is a “do nothing”” statement

```
>>> x = "fried goldfish"
>>> if x == "same for dinner":
...    print "I will destroy the universe"
... else:
...    # I'm fine with that.  I'll do nothing
...    pass
...
>>>
```

```
###  PYTHON BOOT CAMP EXAMPLE;
###  created by Josh Bloom at UC Berkeley, 2012 (ucbpythonclass+bootcamp@gmail.com)
###  all rights reserved 2012 (c)
###  https://github.com/profjsb/python-bootcamp

# set some initial variables. Set the initial temperature low
faren = -1000

# we dont want this going on forever, let's make sure we cannot have too many attempts
max_attempts = 6
attempt = 0

while faren < 100:
    # let's get the user to tell us what temperature it is
    newfaren = float(raw_input("Enter the temperature (in Fahrenheit): "))
    if newfaren > faren:
            print "It's getting hotter"
    elif newfaren < faren:
            print "It's getting cooler"
    else:
        # nothing has changed, just continue in the loop
        continue
    faren = newfaren # now set the current temp to the new temp just entered
    attempt += 1 # bump up the attempt number
    if attempt >= max_attempts:
        # we have to bail out
        break

if attempt >= max_attempts:
    # we bailed out because of too many attempts
    print "Too many attempts at raising the temperature."
else:
    # we got here because it's hot
    print "it's hot here, man."
```

File: temp1.py

```
BootCamp> python temp1.py
Enter the temperature (in Fahrenheit): 1
It's getting hotter
Enter the temperature (in Fahrenheit): 2
It's getting hotter
Enter the temperature (in Fahrenheit): 3
It's getting hotter
Enter the temperature (in Fahrenheit): 4
It's getting hotter
Enter the temperature (in Fahrenheit): -1
It's getting cooler
Enter the temperature (in Fahrenheit): 10
It's getting hotter
Too many attempts at raising the temperature.
BootCamp>
```

```
BootCamp> python temp1.py
Enter the temperature (in Fahrenheit): 3
It's getting hotter
Enter the temperature (in Fahrenheit): -45
It's getting cooler
Enter the temperature (in Fahrenheit): 101
It's getting hotter
it's hot here, man.
BootCamp>
```

```
###  PYTHON BOOT CAMP EXAMPLE;
###  created by Josh Bloom at UC Berkeley, 2012 (ucbpythonclass+bootcamp@gmail.com)
###  all rights reserved 2012 (c)
###  https://github.com/profjsb/python-bootcamp

# set some initial variables. Set the initial temperature low
faren = -1000

# we dont want this going on forever, let's make sure we cannot have too many attempts
max_attempts = 6
attempt = 0

while faren < 100 and (attempt < max_attempts):
    # let's get the user to tell us what temperature it is
    newfaren = float(raw_input("Enter the temperature (in Fahrenheit): "))
    if newfaren > faren:
            print "It's getting hotter"
    elif newfaren < faren:
            print "It's getting cooler"
    else:
        # nothing has changed, just continue in the loop
        continue

    faren = newfaren
    attempt += 1    # bump up the attempt number

if attempt >= max_attempts:
    # we bailed out because of too many attempts
    print "Too many attempts at raising the temperature."
else:
    # we got here because it's hot
    print "it's hot here, man."
```

File: temp2.py

Exercise for the Breakout

Write a program which allows the user to build up a sentence one word at a time, stopping when they enter a period (.), exclamation (!), or question mark (?).

Example interaction:

```
Please enter a word in the sentence (enter . ! or ? to end.): My
...currently: My
Please enter a word in the sentence (enter . ! or ? to end.): name
...currently: My name
Please enter a word in the sentence (enter . ! or ? to end.): is
...currently: My name is
Please enter a word in the sentence (enter . ! or ? to end.): Slim
...currently: My name is Slim
Please enter a word in the sentence (enter . ! or ? to end.): Shady
...currently: My name is Slim Shady
Please enter a word in the sentence (enter . ! or ? to end.): !
--->My name is Slim Shady!
```