TEEX CYBER INNOVATION CAMP
2017

Python: The Basics

Python Origin

TEEX CYBER INNOVATION CAMP 2017

- Created by **Guido Van Rossum** in early 1990s
  - Conceived at late 1980’s
  - Named after “Monty Python’s Flying Circus”

- Short History of Python Versions
  - Python 1.0 – January 1994
  - Python 2.0 – October 2000
    - Python 2.6 – October 2008
    - Python 2.7 – July 2010 (latest version)
  - Python 3.0 – December 2008
    - NOTE: Mostly compatible with Python 2

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
    y = y + “ World” # String concatenation
print x
print y
```

Enough to Understand the Code

TEEX CYBER INNOVATION CAMP 2017

Assignment uses = and comparison uses ==.

For numbers + - * / % are as expected.

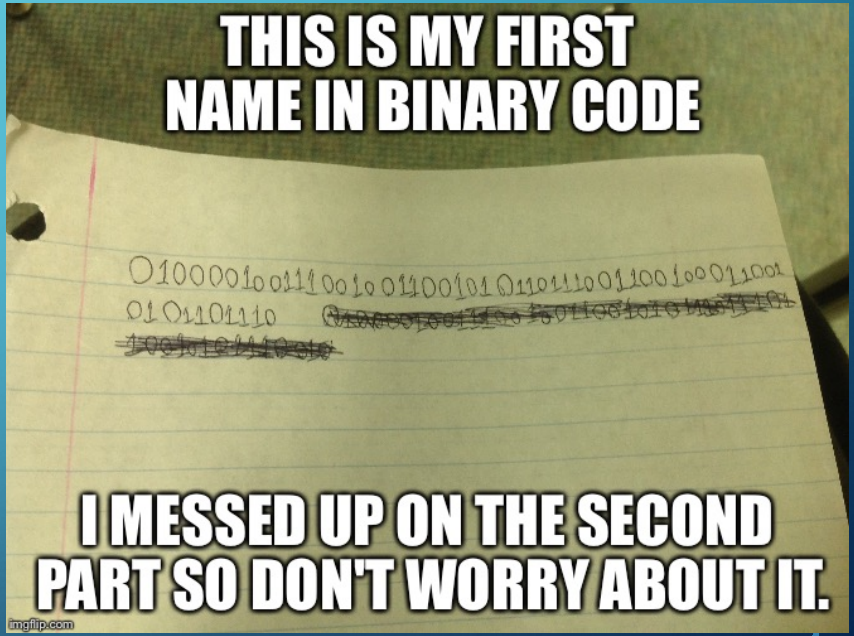
- Special use of + for string concatenation.
- Special use of % for string formatting (as with printf in C)

Logical operators are words (and, or, not) *not* symbols

The basic printing command 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.


THIS IS MY FIRST NAME IN BINARY CODE
0100001001110010011001010110111001100100011001
01 01101110
I MESSED UP ON THE SECOND PART SO DON'T WORRY ABOUT IT.
imgflip.com

Basic Datatypes

- Integers

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

- Floats

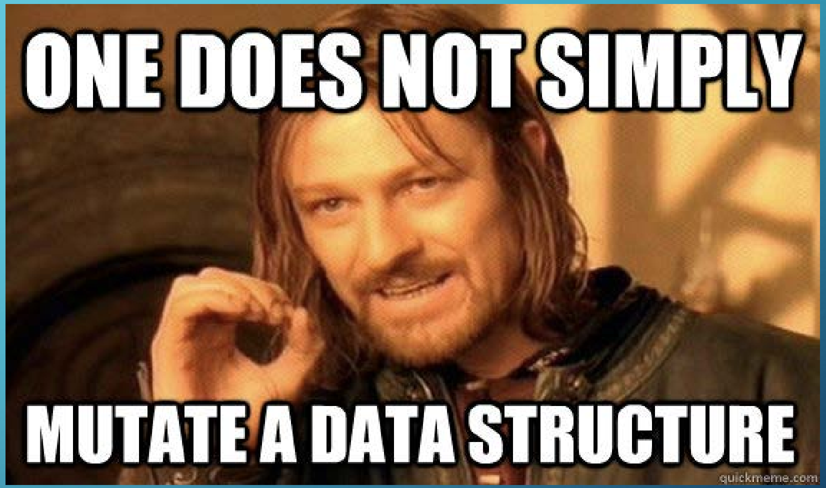
```
x = 3.456
```

- Strings
  - Can use “” or ‘’ to specify.

    ```
    "abc" 'abc'
    ```
    (Same thing.)
  - Unmatched can occur within the string.

    ```
    "matt's"
    ```
  - Use triple double-quotes for multi-line strings or strings than contain both ' and " inside of them:

    ```
    """a'b"c"""
    ```



TEEX CYBER INNOVATION CAMP 2017

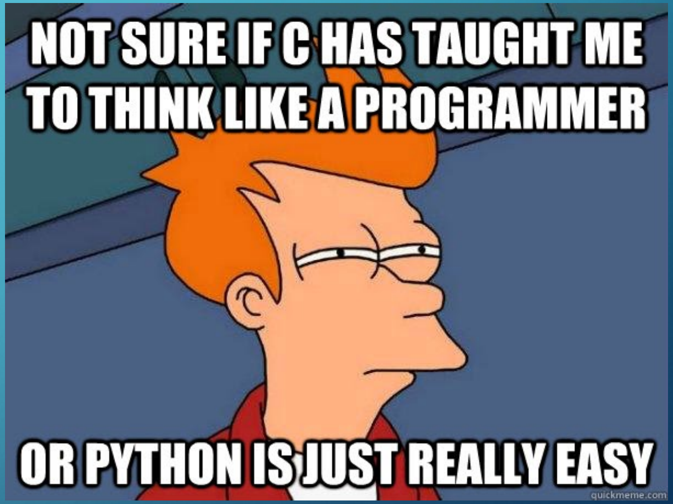
Printing to screen

Python 2: print x

```
print 6
print 5*4
print tiger     #prints the value of the tiger variable
```

Python 3: print(x)

```
print 6
print 5*4
print tiger     #prints the value of the tiger variable
```


NOT SURE IF C HAS TAUGHT ME TO THINK LIKE A PROGRAMMER
OR PYTHON IS JUST REALLY EASY
quickmeme.com

Whitespace

- **Whitespace 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.

- **Use *consistent* indentation to mark blocks of code .**

  The first line with *less* indentation is outside of 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.)**

Comments

- Start comments with # – the rest of line is ignored.

Variables and Assignments

- Variable names are case-sensitive, contain letters, numbers or _, cannot start with a number
- Variables do not have an intrinsic type
- Variables must be assigned before they can be referenced
- Variable on the left side, value on right side of = sign

```
temperature = 98.6
myName = "Monique"
x, y = 0, 100
```

Reserved Words

You can't use some key words as variables (because they're used by Python for other things)

```
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
```

Simple Types

- Numbers, strings and tuples are simple types (also called *immutable*). If you assign a variable with a simple type to another variable, Python makes a copy of the value and puts it into the new variable

Example:

```
x = 3
y = x
y = y + 1     # sets y to 4, value of x still 3
```

More Complicated Types

- Lists, objects (also called *mutable*), when you assign one variable to another, both variables end up pointing to the same values in memory

List example:

```
myList = ["John", "Paul", "George", "Ringo"]
yourList = myList    # this makes both variables point to the same value
yourList = ["Yoko","John"]    # myList now also points to ["Yoko","John"]
```

Lists and Tuples (part 1)

Similar to arrays, but every time you assign a Tuple to a variable it makes a new copy. Assigning a List to a variable makes the variable "point" to the list's value. **List elements can change. Tuple elements cannot.**

Here's how to assign them to variables:

```
someList = [10,20,30,40]

someTuple = (10,20,30,40)
```

Lists and Tuples (part 2)

To reference one element in a list or tuple (or one character in a string!), put it in square brackets:

```
years = ["freshman","sophomore","junior","senior"]
print years[0]     # prints "freshman" (without the quotes)
```

Changing a list element

```
years[0] = "fish"
print years[0]     # prints "fish" (without the quotes)
```

Lists and Tuples (part 3)

Lists and tuples can contain all different types of elements

Example:

```
historyClass = ("HIST","Early American Culture",93, 4)
```

Lists and Tuples (part 4)

- Indexing starts at 0 (first element). Negative indexing starts at -1 (last element)

```
cities = ["College Station","Bryan","Brenham","Iola","North Zulch","Snook"]
cities[0]      # "College Station"
cities[4]      # "North Zulch"
cities[-2]     # "North Zulch"
```

Lists and Tuples (part 5)

Slicing up lists, tuples and strings.

```
slogan = "Don't mess with Texas"
slogan[16:21]        # "Texas"
slogan[16:-1]        # "Texas"
slogan[11:]          # "with Texas"
slogan[:5]           # "Don't"
slogan[:]            # "Don't mess with Texas"
```

True or False (*Boolean*)

Operators: `==,  !=,  <,  <=,  >,  >=,  in`

Examples:

```
x, y, name = 1, 4, "Harry"
x == y         # False
x < y          # True
'r' in name    # True
```

+ Operator

Concatenates lists, tuples and strings together

```
(1, 2, 3) + (4, 5, 6)    # (1, 2, 3, 4, 5, 6)
[1 ,2 ,3] + [4, 5, 6]    # [1, 2, 3, 4, 5, 6]
"I like" + " " + "Python"  # "I like Python"
```

* Operator

Repeats multiples of lists, tuples and strings

```
(1, 2, 3) * 3    # (1, 2, 3, 1, 2, 3, 1, 2, 3)
[4, 5, 6] * 2    # [4, 5, 6, 4, 5, 6]
"Python" * 4     # "PythonPythonPythonPython"
```

Main Differences between Python 2 and 3

- Input

  Python 2: `raw_input("Enter a number:")`

  Python 3: `input("Enter a number:")`

- Print

  Python 2: **`Print x`** Python 3: **`Print(x)`**

- Integer division

  Python 2: `3/2     #1   (int/int -> int)`

  Python 3: `3/2     #1.5 (int/int -> real)`

- Loop variables global (Python2) changed to local (Python3)

List method: append

```
friends = ["Al","Bev"]
friends.append("Chuck") #friends=["Al","Bev","Chuck"]
friends.append(1,"Chuck") #friends=["Al","Chuck","Bev"]
```

append is a *method* of lists

List method: extend

```
friends = ["Al","Bev"]

moreFriends = ["Chuck","Dee"]

friends.append(moreFriends) #friends=["Al","Bev",["Chuck","Dee"]]

friends.extend(moreFriends) #friends=["Al","Bev","Chuck","Dee"]
```

extend is another *method* of lists

+ creates a new list, extend modifies current list

List methods: index, count, remove, reverse, sort

```
friends = ["Al","Bev","Chuck","Dee","Bev"]

friends.index("Dee") # 3

friends.count("Bev") # 2

friends.remove("Bev") # removes first instance only: ["Al","Chuck","Dee","Bev"]

friends.reverse() # ["Bev","Dee","Chuck","Al"]

friends.sort() # removes first instance only: ["Al","Bev","Chuck","Dee"]
```

Convert tuple to list and back

```
x = [1,2,3]
y = tuple(x)   # y = (1,2,3)
z = list( (5,6,7,8,9) )  # z = [5,6,7,8,9]
```

Dictionaries

- Dictionaries are like "Super Lists", where instead of referring to a ordered integer index (0,1,2,3... etc) , the key can be any simple type

Examples:

```
class= {'subject':'Math','days':'MTWHF','nStudents':100}
class[nStudents]    # 100
class['subject'] = "Mathematics"   #changes 'Math' to 'Mathematics'
class.keys()    # ['subject','days','nStudents']
class.values()     # ['Mathematics','MTWHF',100]
```

Functions

Functions are like sub-processes that are executed by using just their name and the values they should use

```
def add1(num):
    return num + 1



def makeClass(subj, whenTaught, howMany):
    class = {'subject':subj, 'days':whenTaught, 'nStudents':howMany}
    return class
```

Function arguments

The order of the values is important:

```
def y(slope,x,yIntercept):
    return slope * x + yIntercept

yValue = y(4,2,10)    # slope is 4, x is 2, yIntercept is 10
                      # yValue gets set to 18
```

Optional Arguments to Functions

Easier shown than described

```
def add(a,b=0, c=0, d=0, e=0):
    return a + b + c + d + e


add(10)        # 10
add(1,2,3)     # 6
```

Functions (continued): tricky bits

All functions return some value

If no `return` statement, the value returned is ***`None`***

All functions must have different names

Program control, or connecting it all

Statements, loops, blocks

Example:

```
if x == 3:
    print "X equals 3."
elif x == 2:
    print "X equals 2."
else:
    print "X equals something else."
print "This is outside the 'if'."
```

While loop

Keep doing something as long as a condition is met

Example:

```
x = 1
while x < 5:
    print(x)
    x += 1    # very important! Must eventually make the condition False!
print("All Done")
```

For loop

Execute the interior of the loop a certain number of times

```
for x in range(5):
    print(x)
print("Done")
```

If…elif…else…

Example:

```
x = input("Enter a fruit:")
if (x == "Orange"):
    print ("My color is orange")
elif (x == "Banana"):
    print ("I like yellow")
else:
    print ("It's not orange or yellow")
```

…and Lots More!

Python is a powerful language for many, many applications today! It includes elements of object-oriented programming (OOP), libraries, data hiding.

Most of these features are left for you to explore!