



**Please make sure you have python 2.7.x on your machines.**

**No Lower or higher versions please.**

You can download it from [python.org](http://python.org)



# **An Introduction to Python**

Ramesh S

# What is Python?

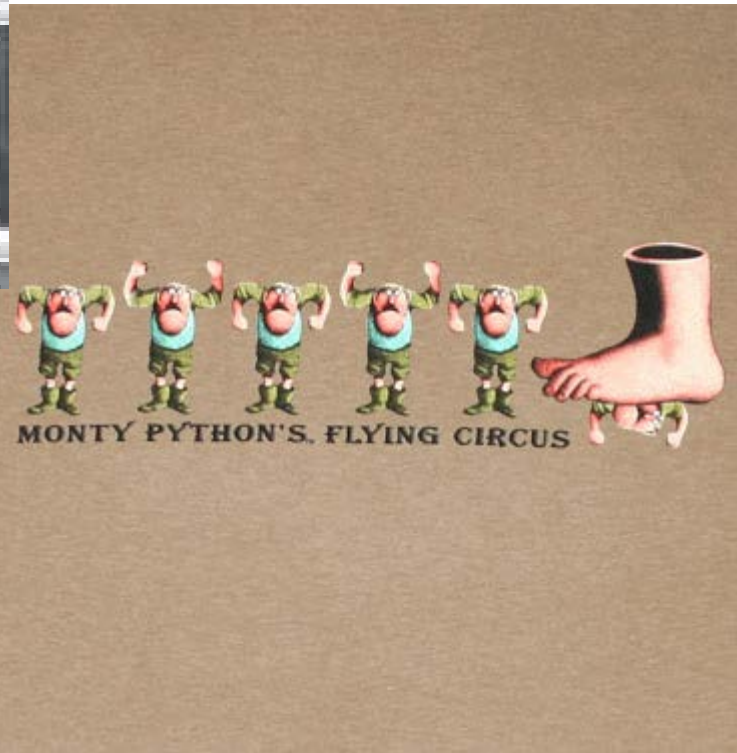
- Python is a high level programming language created by [Guido van Rossum](#).
  - Developed at [National Research Institute for Mathematics and Computer Science](#) in the [Netherlands](#).
  - Currently maintained by [Python Software Foundation](#).
  - Named after the British comedy troupe, [Monty Python's Flying Circus](#).
  - First released in 1991
- 



Guido van Rossum

Benevolent Dictator for Life (BDFL)  
- Python Software Foundation.

# Monty Python Flying Circus



# Python Release History

- Python 3.3.2 (May 2013)
  - Python 3.2.5 (May 2013)
  - Python 3.1.5 (April 2012)
  - Python 3.0.1 (February 2009)
  - Python 2.7.5 (May 2013)
  - Python 2.6.8 (April 2012)
  - Python 2.5.6 (May 2011)
  - Python 2.4.6 (Dec 2008)
  - Python 2.3.7 (March 2008)
  - Python 2.2.3 (May 2003)
  - Python 2.1.3 (April 2002)
  - Python 2.0.1 (June 2001)
  - Python 1.6.1 (Sep 2000)
  - Python 1.5.2 (April 1999)
- 

# What is Python?

Python is an

- easy to learn,
- powerful programming language.

It has efficient

- high-level data structures
- and a simple but effective approach to
- object-oriented programming.


Python's

- elegant syntax and
- dynamic typing,

together with its

interpreted nature,

make it an ideal language for

- scripting and
  - rapid application development
  - in many areas on most platforms.
- 



# Design Goals

- Designed to be simple yet powerful
- Allow modular programming
- Great emphasis on readability
- Rapid application development
- Easy to embed in & extend with other languages






# Philosophy of Python

- import this




# Why Python?


So, why use Python over other scripting languages?

- Well tested and widely used
  - Great documentation
  - Strong community support
  - Widely Cross-platform
  - Open sourced
  - Can be freely used and distributed, even for commercial purposes.
- 

# Where is Python used

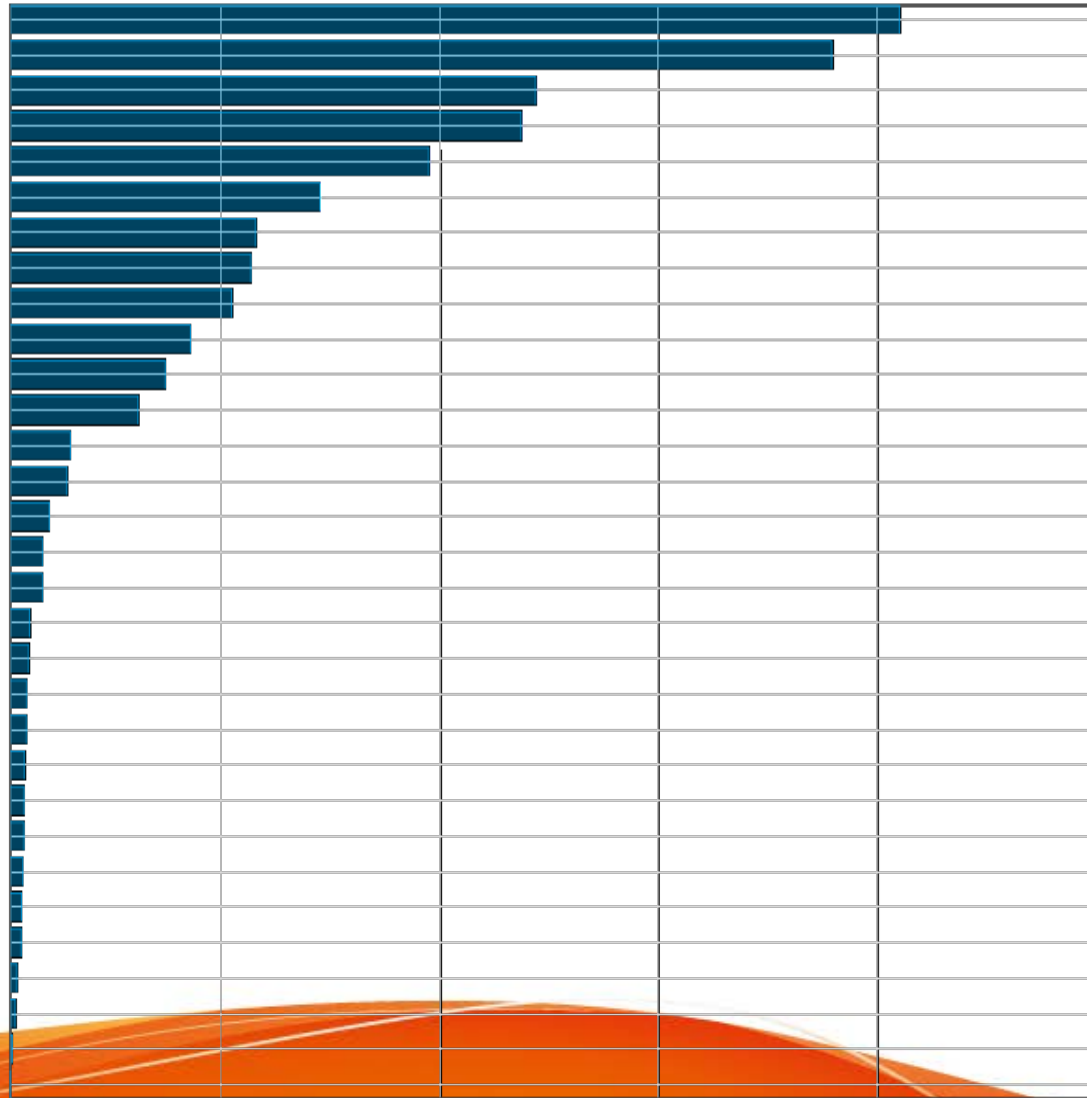
- Scripting
  - Rapid Prototyping
  - Text Processing
  - Web applications
  - GUI programs
  - Game Development
  - Database Applications
  - System Administration Automation
- 

# Who is using Python

- Used by large organizations
    - NASA
    - Google
    - Microsoft
    - eBay
    - PayPal
    - Dropbox
    - OpenStack
  - Used in published games
    - Battlefield 2
    - Civilization IV
    - Disney's Toontown Online
- 

# How popular is Python

Java  
C  
C++  
Javascript  
PHP  
Python.....  
SQL  
Perl  
C#



# Learning Python is Easy - Check these out



What does this program do?

```
print "Hello World"
```





# What does this program do?

```
count=0
```

```
while count<11:
```

```
    print count
```

```
    count+=1
```



# What does this program do?

```
kids=['Lahari','Lalitha','Jithu','Vishal','Ujwal','Nitesh']  
for kid in kids:  
    print kid
```



# What does this program do?

```
kids=['Lahari','Lalitha','Jithu','Vishal','Ujwal','Nitesh']  
for kid in kids:  
    if len(kid)==6:  
        print kid
```



# What does this program do?

```
kids=['Lahari','Lalitha','Jithu','Vishal','Ujwal','Nitesh']  
for kid in kids:  
    if "it" in kid:  
        print kid
```



# What does this program do?

```
marks={'maths':97,'science':'98','history':78}  
for subject,marks in marks.items():  
    print subject,marks
```



# What does this program do?

```
f=open('dictionary.txt','r')
```

```
for line in f:
```

```
    if line.startswith('s'):
```

```
        print line
```



# What does this program do?

```
f=open('words.txt','r')
```

```
j=open('new-words.txt','w')
```

```
j.write(f.read())
```

```
f.close()
```

```
j.close()
```





# What does this program do?


```
def factorial(n):  
    fact=1  
    for x in range(n,1,-1):  
        fact*=x  
    return fact  
  
print factorial(4)
```

What does this program do?

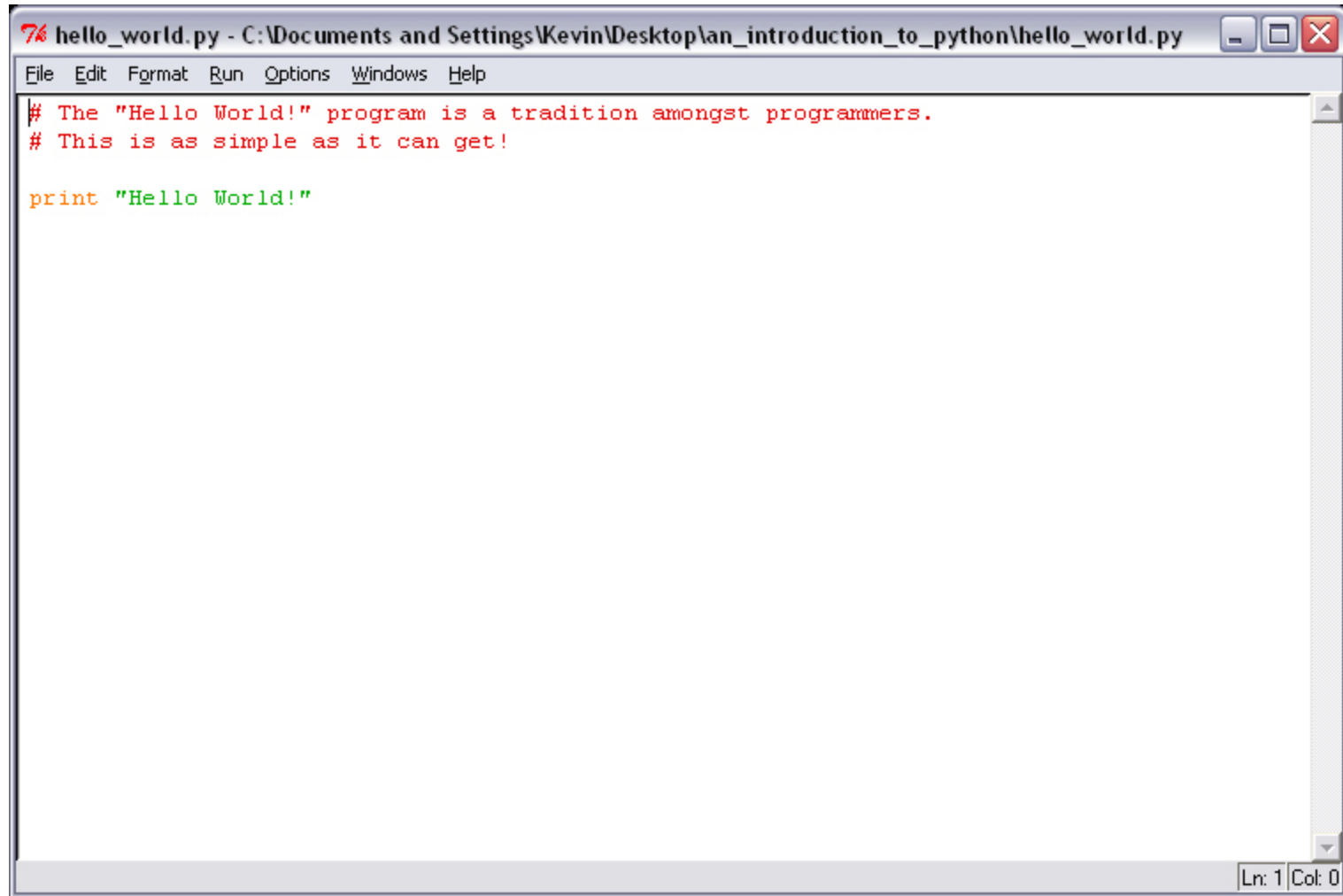
```
squares=[]  
for x in range(1,100):  
    squares.append(x*x)  
  
print squares
```



# How do I edit Python?

- IDLE is a free GUI based text editor which ships with Python.
  - You could just write your scripts with Notepad, but IDLE has many helpful features like keyword highlighting.
  - Click your “Start” button and Find “Python 3.1” on your “All Programs” menu. Then select and run “IDLE (Python GUI)”.
  - You can also right-click on any python script (.py) and select “Edit with IDLE” to open it.
- 

# IDLE in Action



The screenshot shows the Python IDLE (Integrated Development and Learning Environment) window. The title bar indicates the file is 'hello\_world.py' located at 'C:\Documents and Settings\Kevin\Desktop\an\_introduction\_to\_python\hello\_world.py'. The menu bar includes 'File', 'Edit', 'Format', 'Run', 'Options', 'Windows', and 'Help'. The main text area contains the following Python code:


```
# The "Hello World!" program is a tradition amongst programmers.  
# This is as simple as it can get!  
  
print "Hello World!"
```

The status bar at the bottom right shows 'Ln: 1 Col: 0'.

# “Hello World!”

- The “Hello World!” example is a tradition amongst programmers, so lets start there...
- Run IDLE and type in the following

```
print( “Hello World!” )
```

- Save your work and hit F5 to run it.
  - The script simply prints the string, “Hello World!” to the console.
- 


# The print function

- The print function simply outputs a string value to our script's console.

```
print( "Hello World!" )
```

- It's very useful for communicating with users or outputting important information.

```
print( "Game Over!" )
```

- We get this function for free from Python.
  - Later, we'll learn to write our own functions.
- 

# Adding Comments

- Comments allow us to add useful information to our scripts, which the Python interpreter will ignore completely.
- Each line for a comment must begin with the number sign '#'.

```
# This is a programming tradition...  
print ( "Hello World!" )
```

- Do yourself a favor and use them!





# Quoting Strings

- Character strings are denoted using quotes.
- You can use double quotes...

```
print( "Hello World!" ) # Fine.
```

- Or, you can use single quotes...

```
print( 'Hello World!' ) # Also fine.
```

- It doesn't matter as long as you don't mix them


```
print( "Hello World!" ) # Syntax Error!
```



# Triple Quotes

- You can even use triple quotes, which make quoting multi-line strings easier.

```
print( """  
This is line one.  
This is line two.  
This is line three.  
Etc...  
""" )
```



# Strings and Control Characters

- There are several control characters which allow us to modify or change the way character strings get printed by the `print` function.
- They're composed of a backslash followed by a character. Here are a few of the more important ones:

`\n` : New line

`\t` : Tabs

`\\` : Backslash

`\'` : Single Quote

`\"` : Double Quote




# Variables

- Like all programming languages, Python variables are similar to variables in Algebra.
- They act as place holders or symbolic representations for a value, which may or may not change over time.
- Here are six of the most important variable types in Python:

<code>int</code>	Plain Integers	( 25 )
<code>long</code>	Long Integers	( 4294967296 )
<code>float</code>	Floating:point Numbers	( 3.14159265358979 )
<code>bool</code>	Booleans	( True, False )
<code>str</code>	Strings	( "Hello World!" )
<code>list</code>	Sequenced List	( [25, 50, 75, 100] )

# Type:less Variables

- Even though it supports variable types, Python is actually type:less, meaning you do not have to specify the variable's type to use it.
  - You can use a variable as a character string one moment and then overwrite its value with a integer the next.
  - This makes it easier to learn and use the language but being type:less opens the door to some hard to find bugs if you're not careful.
  - If you're uncertain of a variable's type, use the `type` function to verify its type.
- 

# Rules for Naming Variables

- You can use letters, digits, and underscores when naming your variables.
- But, you cannot start with a digit.

var = 0 # Fine.

var1 = 0 # Fine.

var\_1 = 0 # Fine.

\_var = 0 # Fine.

1var = 0 # Syntax Error!

# Rules for Naming Variables


- Also, you can't name your variables after any of Python's reserved keywords.

and, del, for, is, raise, assert, elif, from,  
lambda, return, break, else, global, not, try,  
class, except, if, or, while, continue, exec,  
import, pass, yield, def, finally, in, print





# Numerical Precision

- Integers
    - Generally 32 signed bits of precision
    - [2,147,483,647 .. -2,147,483,648]
    - or basically ( $-2^{32}$  ,  $2^{32}$ )
    - Example: 25
  - Long Integers
    - Unlimited precision or size
    - Format: <number>L
    - Example: 4294967296L
  - Floating:point
    - Platform dependant “double” precision
    - Example: 3.141592653589793
- 

# Type Conversion

- The special constructor functions `int`, `long`, `float`, `complex`, and `bool` can be used to produce numbers of a specific type.
- For example, if you have a variable that is being used as a float, but you want to use it like an integer do this:

```
myFloat = 25.12  
myInt = 25  
print( myInt + int( myFloat ) )
```

- With out the explicit conversion, Python will automatically upgrade your addition to floating-point addition, which you may not want especially if your intention was to drop the decimal places.



# Type Conversion

- There is also a special constructor function called `str` that converts numerical types into strings.

```
myInt = 25  
myString = "The value of 'myInt' is "  
print( myString + str( myInt ) )
```

- You will use this function a lot when debugging!
- Note how the addition operator was used to join the two strings together as one.



# Arithmetic Operators

- Arithmetic Operators allow us to perform mathematical operations on two variables or values.
- Each operator returns the result of the specified operation.

+ Addition

- Subtraction

\* Multiplication

/ Float Division

// Floor Division

\*\* Exponent


abs Absolute Value



# Comparison Operators

- Comparison Operators return a True or False value for the two variables or values being compared.
  - < Less than
  - <= Less than or equal to
  - > Greater than
  - >= Greater than or equal to
  - == Is equal to
  - != Is not equal to

# Operator Precedence

- lambda Lambda Expression
  - if - else Conditional expression
  - or Boolean OR
  - and Boolean AND
  - not x Boolean NOT
  - in, not in, is, is not, <, <=, >, >=, !=, == Comparisons, including
  - membership tests and identity tests
  - | Bitwise OR
  - ^ Bitwise XOR
  - & Bitwise AND
  - <<, >> Shifts
  - +, - Addition and subtraction
  - \*, /, //, % Multiplication, Division, Floor Division and Remainder
  - +x, -x, ~x Positive, Negative, bitwise NOT
  - \*\* Exponentiation
  - x[index], x[index-index], x(arguments...), x.attribute Subscription,
  - slicing, call, attribute reference
  - (expressions...), [expressions...], {key- value...}, {expressions...}
  - Binding or tuple display, list display, dictionary display, set display
- 

# Boolean Operators

- Python also supports three Boolean Operators, **and**, **or**, and **not**, which allow us to make use of Boolean Logic in our scripts.
- Below are the Truth Tables for **and**, **or**, and **not**.

**and**

InA	InB	Out
0	0	0
0	1	0
1	0	0
1	1	1

**or**

InA	InB	Out
0	0	0
0	1	1
1	0	1
1	1	1

**not**

In	Out
0	1
1	0

# Boolean Operators

Suppose that... `var1 = 10`.

- The `and` operator will return True if and only if both comparisons return True.


```
print( var1 == 10 and var1 < 5 )    (Prints False)
```

- The `or` operator will return True if either of the comparisons return True.

```
print( var1 == 20 or var1 > 5 )      (Prints True)
```

- And the `not` operator simply negates or inverts the comparison's result.

```
print( not var1 == 10 )              (Prints False)
```





# Special String Operators

- It may seem odd but Python even supports a few operators for strings.
- Two strings can be joined (concatenation) using the + operator.

```
print( "Game " + "Over!" )
```

Outputs "Game Over!" to the console.


- A string can be repeated (repetition) by using the \* operator.

```
print( "Bang! " * 3 )
```


Outputs "Bang! Bang! Bang! " to the console.




# Lists

- Dynamically resizable
  - Indexable
  - Slicable
  - `append()`, `insert()`, `extend()`
  - `pop()`, `remove()`
  - `index()`, `count()`
  - `sort()`, `reverse()`
  - `'in'` operator for presence check
- 


# Tuple

- Immutable list
  - Indexable
  - Slicable
  - `index()`, `count()`
  - `'in'` operator for presence check
- 


# Dictionary

- key - value pairs
  - key required to access a value
  - keys must be unique
  - values need not be
  - insertion order not maintained
  - `keys()`, `values()`, `items()`
  - `get()`, `has_key()`
  - `pop()`, `popitem()`
- 

# Set

- unordered collection of unique items
  - union()
  - intersection()
  - '-' operator supported
  - add()
  - remove()
- 

# Flow Control

- Flow Control allows a program or script to alter its flow of execution based on some condition or test.
  - The most important keywords for performing Flow Control in Python are **if**, **else**, **elif**, **for**, and **while**.
- 

# If Statement

- The most basic form of Flow Control is the `if` statement.

`# If the player's health is less than or equal to 0 - kill him!`

`if health <= 0:`

`print( "You're dead!" )`

- Note how the action to be taken by the `if` statement is indented or tabbed over. This is not a style issue – it's required.
- Also, note how the `if` statement ends with a semi-colon.



# If-else Statement

- The **if-else** statement allows us to pick one of two possible actions instead of a all-or-nothing choice.

```
health = 75
```

```
if health <= 0:  
    print( "You're dead!" )  
else:  
    print( "You're alive!" )
```

- Again, note how the **if** and **else** keywords and their actions are indented. It's very important to get this right!




# If-elif-else Statement

- The **if-elif-else** statement allows us to pick one of several possible actions by chaining two or more **if** statements together.


```
health = 24
```

```
if health <= 0:  
    print( "You're dead!")  
elif health < 25:  
    print( "You're alive - but badly wounded!")  
else:  
    print( "You're alive!")
```



# Multiple elif

```
if hour == 0:  
    # midnight  
elif hour == 6:  
    # morning  
elif hour == 17:  
    # evening  
else:  
    print "Good Day!"
```



# while Statement

- The **while** statement allows us to continuously repeat an action until some condition is satisfied.

```
numRocketsToFire = 3  
rocketCount = 0
```

```
while rocketCount < numRocketsToFire:  
    # Increase the rocket counter by one  
    rocketCount = rocketCount + 1  
    print( "Firing rocket #" + str( rocketCount ) )
```

- This **while** statement will continue to 'loop' and print out a "Firing rocket" message until the variable "rocketCount" reaches the current value of "numRocketsToFire", which is 3.


# for Statement

- The **for** statement allows us to repeat an action based on the iteration of a Sequenced List.

```
weapons = [ "Pistol", "Rifle", "Grenade", "Rocket Launcher" ]
```

```
print "-- Weapon Inventory --"
```

```
for x in weapons:  
    print( x)
```

- The **for** statement will loop once for every item in the list.
  - Note how we use the temporary variable 'x' to represent the current item being worked with.
- 

# break Keyword

- The **break** keyword can be used to escape from **while** and **for** loops early.

```
numbers = [100, 25, 125, 50, 150, 75, 175]
```

```
for x in numbers:  
    print( x )  
    # As soon as we find 50 - stop the search!  
    if x == 50:  
        print( "Found It!" )  
        break;
```

- Instead of examining every list entry in “numbers”, The **for** loop above will be terminated as soon as the value 50 is found.

# continue Keyword

- The **continue** keyword can be used to short-circuit or bypass parts of a **while** or **for** loop.

```
numbers = [100, 25, 125, 50, 150, 75, 175]
```

```
for x in numbers:  
    # Skip all triple digit numbers  
    if x >= 100:  
        continue;  
    print( x )
```

- The **for** loop above only wants to print double digit numbers. It will simply **continue** on to the next iteration of the **for** loop if x is found to be a triple digit number.

# Functions


- A function allows several Python statements to be grouped together so they can be called or executed repeatedly from somewhere else in the script.
- We use the `def` keyword to define a new function.



# Defining functions


- Below, we define a new function called “printGameOver”, which simply prints out, “Game Over!”.

```
def printGameOver():  
    print( “Game Over!” )
```

- Again, note how indenting is used to denote the function’s body and how a semi-colon is used to terminate the function’s definition.
- 



# Function arguments

- Often functions are required to perform some task based on information passed in by the user.
  - These bits of Information are passed in using function arguments.
  - Function arguments are defined within the parentheses “()”, which are placed at the end of the function’s name.
- 

# Function arguments

- Our new version of `printGameOver`, can now print out customizable, “Game Over!”, messages by using our new argument called “playersName”.

```
def printGameOver( playersName ):  
    print( “Game Over... ” + playersName + “!” )
```

- Now, when we call our function we can specify which player is being killed off.



# Default Argument Values

- Once you add arguments to your function, it will be illegal to call that function without passing the required arguments.
- The only way around this is to specify a default argument value.



# Default Argument Values

- Below, the argument called “playersName” has been assigned the default value of, “Guest”.

```
def printGameOver( playersName="Guest" ):
    print( "Game Over... " + playersName + "!" )
```

- If we call the function and pass nothing, the string value of “Guest” will be used automatically.



# Multiple Function arguments

- If we want, we can define as many arguments as we like.

```
def printGameOver( playerName, totalKills ):  
    print( "Game Over... " + playerName + "!" )  
    print( "Total Kills: " + str( totalKills ) + "\n" )
```

- Of course, we must make sure that we pass the arguments in the correct order or Python will get confused.

```
printGameOver( "camper_boy", 15 ) # Correct!  
printGameOver( 12, "killaHurtz" ) # Syntax Error!
```

# Keyword arguments

- If you really want to change the order of how the arguments get passed, you can use keyword arguments.

```
def printGameOver( playerName, totalKills ):  
    print( "Game Over... " + playerName + "!" )  
    print( "Total Kills: " + str( totalKills ) + "\n" )  
  
printGameOver( totalKills=12, playerName="killaHurtz" )
```

- Note how we use the argument's name as a keyword when calling the function.



# Function Return Values

- A function can also output or return a value based on its work.
- The function below calculates and returns the average of a list of numbers.

```
def average( numberList ):  
    numCount = 0  
    runningTotal = 0  
  
    for n in numberList:  
        numCount = numCount + 1  
        runningTotal = runningTotal + n  
  
    return runningTotal / numCount
```

- Note how the list's average is returned using the **return** keyword.
- 

# Assigning the return value

- Here's how the return value of our `average` function would be used...

```
myNumbers = [5.0, 7.0, 8.0, 2.0]
```

```
theAverage = average( myNumbers )
```

```
print( "The average of the list is " + str( theAverage ) + "." )
```

- Note how we assign the return value of `average` to our variable called "theAverage".





# Multiple Return Values

- A function can also output more than one value during the return.
- This version of `average` not only returns the average of the list passed in but it also returns a counter which tells us how many numbers were in the list that was averaged.

```
def average( numberList ):
    numCount = 0
    runningTotal = 0

    for n in numberList:
        numCount = numCount + 1
        runningTotal = runningTotal + n

    return runningTotal / numCount, numCount
```

- Note how the `return` keyword now returns two values which are separated by a comma.

# Assigning the return values

- Here's how the multiple return value of our `average` function could be used...

```
myNumbers = [5.0, 7.0, 8.0, 2.0]
```

```
theAverage, numberCount = average( myNumbers )
```


```
print( "The average of the list is " + str( theAverage ) + ".“ )
```

```
print( "The list contained " + str( numberCount ) + " numbers.“)
```


- Note how we assign the return values of `average` to our variables “theAverage” and “numberCount”.



# Functions

- Simple Function
  - Function with arguments
  - Function with default arguments
  - Function with multiple returns
  - Calling a function with keyword arguments
  - Variable length parameters using \*
  - Combination of \* and \*\*
- 


# Introspection commands

- id
  - type
  - dir
  - help
- 


# File I/O - common scenarios

<b>Read file contents at once</b> <pre>f=open('input.txt','r') print( f.read()) f.close()</pre>	<b>Read all lines into a list</b> <pre>f=open('input.txt','r') lines= f.readlines() print( lines) f.close()</pre>
<b>Read file char by char</b> <pre>f=open('input.txt','r') print( f.read(1)) f.close()</pre>	<b>Write into a new file</b> <pre>f=open('input.txt','w') f.write("this is line one\n") f.close()</pre>
<b>Read file line by line</b> <pre>f=open('input.txt','r') print( f.readline()) f.close()</pre>	<b>Append to an existing file</b> <pre>f=open('input.txt','a') f.write("this is a new line \n") f.close()</pre>


# File I/O - all commands

- `open()`
  - `close()`
  - `read()`
  - `readline()`
  - `readlines()`
  - `write()`
  - `writelines()`
  - `seek()`
  - `tell()`
  - `flush()`
  - `closed`
- 

# File modes

- r : read
  - w : write
  - a : append
  - U : universal new line support
  - r+ : read & write
  - w+ : write & read
  - a+ : append & read
  - rb : read in binary mode
  - wb : write in binary mode
  - ab : append in binary mode
- 

# Modules

- Generic import
  - Universal import
  - Function import
  - Function import with rename
  - default name space
  - dir
  - help
- 



# Sample Module

## **mymodule.py**

```
name='python'

def add(a,b,c):
    return a+b+c

def sub(a,b):
    return a-b
```

## **myprogram.py**

```
from mymodule import *

print name

print add(2,3,4)

print sub(7,3)
```

# Sample Module

## **mymodule.py**

```
name='python'

def add(a,b,c):
    return a+b+c

def sub(a,b):
    return a-b
print __name__
```

## **myprogram.py**

```
from mymodule import *

print name

print add(2,3,4)

print sub(7,3)
print __name__
```

# Sample Module

## **mymodule.py**

```
name='python'
def add(a,b,c):
    return a+b+c
def sub(a,b):
    return a-b
```

## **myprogram.py**

```
from mymodule import *
```

```
def main():
    print name
    print add(2,3,4)
    print sub(7,3)
```


```
if __name__=='__main__':
    main()
```



# Python Sequences

- The most basic data structure in Python is the sequence. Each element of a sequence is assigned a number - its position, or index. The first index is zero, the second index is one, and so forth.
- 6 built-in types of sequences. most common ones are lists and tuples.
- There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence, and for finding its largest and smallest elements.

# Python Coding Style

- **PEP 8 has emerged as the style guide that most projects adhere to; it promotes a very readable and eye-pleasing coding style. Every Python developer should read it at some point; here are the most important points extracted for you:**
  - **Use 4-space indentation, and no tabs.**
  - **4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read).**
  - **Tabs introduce confusion, and are best left out.**
  - **Wrap lines so that they don't exceed 79 characters.**
  - **This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.**
- 

# Python Coding Style

- **Use blank lines to separate functions and classes, and larger blocks of code inside functions.**
- **When possible, put comments on a line of their own.**
- **Use docstrings.**
- **Use spaces around operators and after commas, but not directly inside bracketing constructs:**  
**`a = f(1, 2) + g(3, 4)`**
- **Name your classes and functions consistently; the convention is to use**
  - **CamelCase for classes**
  - **lower\_case\_with\_underscores for functions and methods.**

# Python Stack

Component	Example
Python Program	HelloWorld.py
Python	Python.exe or python binary
Python ByteCode Interpreter(VM)	Byte Code
Operating System	Windows, Mac, Linux
Hardware	i386, AMD-64, RISC

# Phases of program execution

Phase	Output
Editor/IDE	HelloWorld.py
Lexical Analysis	Tokens
Parsing	Abstract Syntax Tree (AST).
Code Generation	Byte Code
Python ByteCode Interpreter (It is a stack based Virtual Machine)	Native Machine Code



# Questions



# Credits

Kevin Harris

([Codesampler.com](https://codesampler.com))

