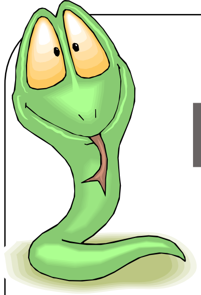
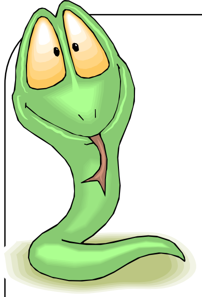
Introduction to Python

Mrs Deepali Vora
Vidyalankar Institute of Technology
Date:02/01/2018



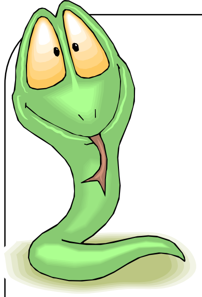
History of Python

- Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.
- Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.
- Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).



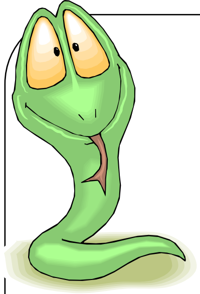
Features of Python

- Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.
- **Python is Interpreted:** Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive:** You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented:** Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language:** Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

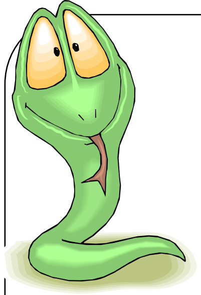


Why Python

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

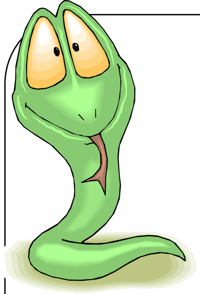


Who uses Python?



Let’s Start - Environment

- Interactive
  - Command Prompt
- GUI / IDE based
  - Basically for Windows
  - Notepad++/Spyder/Jupyter/Anaconda
  - iPython



Basic Datatypes

- Integers (default for numbers)

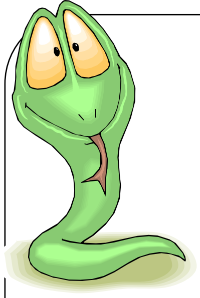
```
z = 5 // 2  # Answer 2, integer division
```

- Floats

```
x = 3.456
```

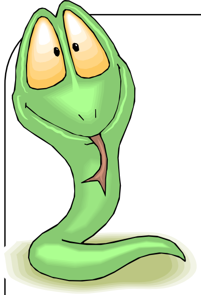
- Strings
  - Can use “” or ‘’ to specify with `“abc” == ‘abc’`
  - Unmatched can occur within the string: `“matt’s”`

- All variables are assigned value by reference.



Enough to Understand the Code

- Indentation matters to code meaning
  - Block structure indicated by indentation
- First assignment to a variable creates it
  - Variable types don’t need to be declared.
  - Python figures out the variable types on its own.
- Assignment is = and comparison is ==
- For numbers + - * / % are as expected
  - Special use of + for string concatenation and % for string formatting (as in C’s printf)
- Logical operators are words (`and, or, not`) *not* symbols
- The basic printing command is `print`



Assignment

- You can assign to multiple names at the same time

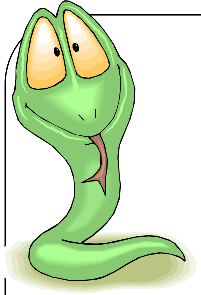
```
>>> x, y = 2, 3
>>> x
2
>>> y
3
```

This makes it easy to swap values

```
>>> x, y = y, x
```

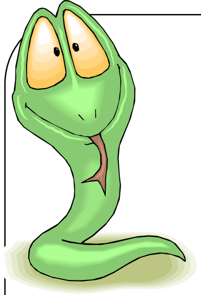
- Assignments can be chained

```
>>> a = b = x = 2
```



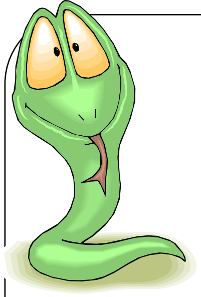
Let’s Try

- Sample1-P1
- Sample1-P2



To read from user

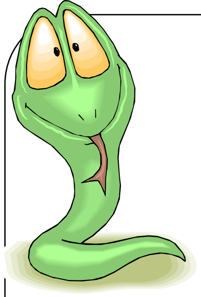
- raw_input() – with python 2.7
- input(msg) – with Python 3.6
- Returns String type of Data
- Conversion functions:
  - int(var1)
  - float(var1)
  - str(var1)
- type(var1) – will give data type of variable



Decision Making

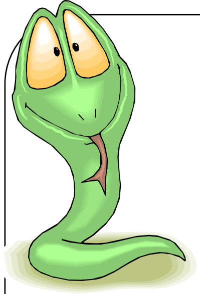
```
if cond :
     Statements
elseif cond:
          statements
else:
     Statements
```

Try:Sample-P3



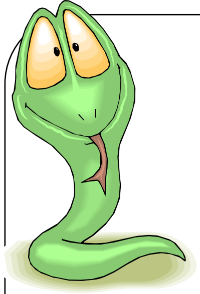
Range Test

- if(3<= a <=5):
- …   print("True")
- … else:
- …   print("False")



Looping

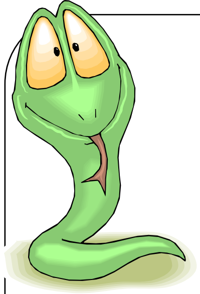
- The `for` statement loops over sequences
-
  ```
  for ch in "Hello":
      print ch
  ```
-
  ```
  H
  e
  l
  l
  o
  ```



Looping

- Built-in function `range()` used to build sequences of integers
-
  ```
  for i in range(3):
     print i
  ```
- `Try - Sample-P11`

  `Sample-P5`
- `Exercise: Print prime numbers in a range`



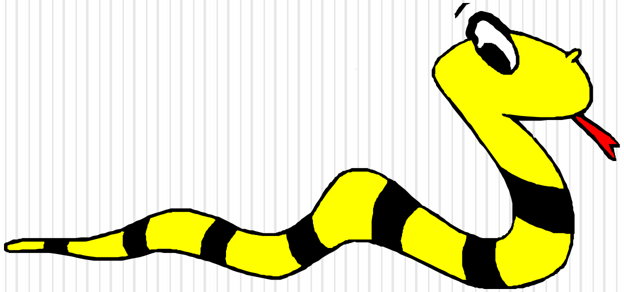
Looping

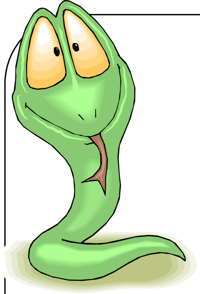
- `while` statement for more traditional loops

-
  ```
  i = 0
  while i < 2:
      print i
      i = i + 1
  ```

- `Try Sample-P6`
- Exercise: Print Fibonacci Series

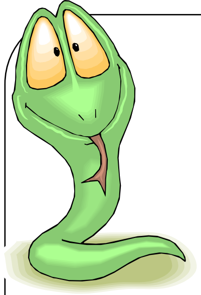
Sequence types: Tuples, Lists, and Strings





Sequence Types

1. Tuple: (‘john’, 32, [CMSC])
    - A simple *immutable* ordered sequence of items
    - Items can be of mixed types, including collection types
2. Strings: “John Smith”
    - *Immutable*
    - Conceptually very much like a tuple
3. List: [1, 2, ‘john’, (‘up’, ‘down’)]
    - *Mutable* ordered sequence of items of mixed types



Similar Syntax

- All three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.
- Key difference:
  - Tuples and strings are *immutable*
  - Lists are *mutable*

Mutable vs Immutable

Immutable

```
>>>x=3
>>>y=x
>>>y=4
>>>print x
3
```

Mutable

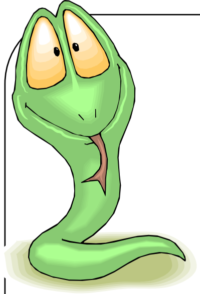
x = some mutable object

y=x

make a change to y

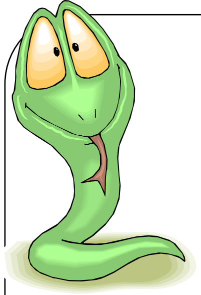
look at x

x will be changed as well



Try this

- a = [1, 2, 3]   # a now references the list [1, 2, 3]
- b = a   # b now references what a references
- a.append(4) # this changes the list a references
- print b   # if we print what b references, [1, 2, 3, 4]

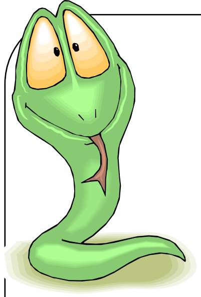


Strings

- S1=“Hello World of Python”
- S2 = “““This is a multi-line string

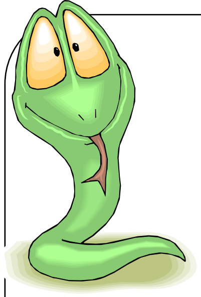
  that uses triple quotes.”””
- Special Operators
  - [ ] ,* ,in , not in, %

- Try StringFun.py

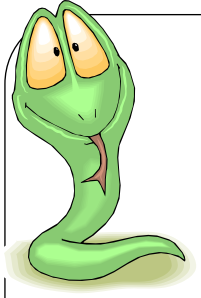


Strings

- Functions
  - len()
  - count(str, beg,end)
  - index(str, beg=0, end=len(string))
  - isalpha()
  - isdigit()
  - islower()
  - isnumeric()
  - isspace()



- Some more….
  - isupper()
  - islower()
  - upper()
  - lower()
  - lstrip
  - rstrip
  - replace(old, new)
  - split(“ch”)
- Try :StringFun2.py



Sequence Types 1

- Define tuples using parentheses and commas

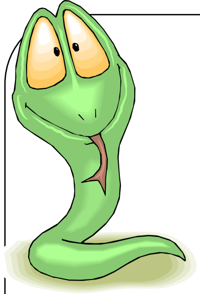
```
>>> tu = (23, ‘abc’, 4.56, (2,3), ‘def’)
```

- Define lists using square brackets and commas

```
>>> li = [“abc”, 34, 4.34, 23]
```

- Define strings using quotes (“, ‘, or “““).

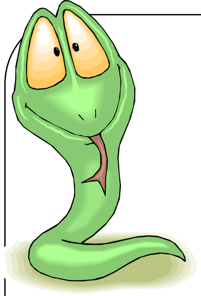
```
>>> st = “Hello World”
>>> st = ‘Hello World’
```



Sequence Types 2

- Access individual members of a tuple, list, or string using square bracket “array” notation
- *Note that all are 0 based…*

```
>>> tu = (23, ‘abc’, 4.56, (2,3), ‘def’)
>>> tu[1]      # Second item in the tuple.
 ‘abc’

>>> li = [“abc”, 34, 4.34, 23]
>>> li[1]       # Second item in the list.
 34

>>> st = “Hello World”
>>> st[1]    # Second character in string.
 ‘e’
```



Positive and negative indices

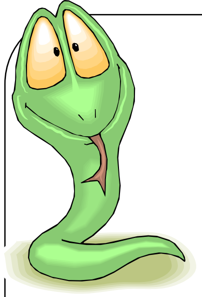
```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0

```
>>> t[1]
'abc'
```

Negative index: count from right, starting with –1

```
>>> t[-3]
4.56
```



Slicing: return copy of a subset

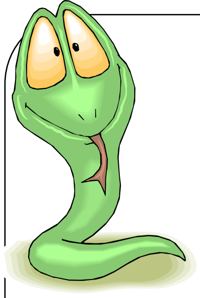
```
>>> t = (23, 'abc', 4.56, (2,3),
'def')
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying *before* second.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

Negative indices count from end

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```



Slicing: return copy of a =subset

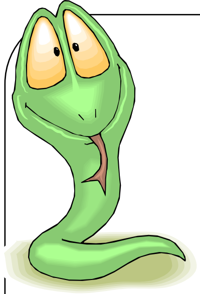
```
>>> t = (23, ‘abc’, 4.56, (2,3),
‘def’)
```

Omit first index to make copy starting from beginning of the container

```
        >>> t[:2]
        (23, ‘abc’)
```

Omit second index to make copy starting at first index and going to end

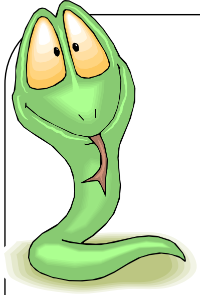
```
        >>> t[2:]
        (4.56, (2,3), ‘def’)
```



Copying the Whole Sequence

- [:] makes a *copy* of an entire sequence

```
>>> t[:]
(23, ‘abc’, 4.56, (2,3), ‘def’)
```



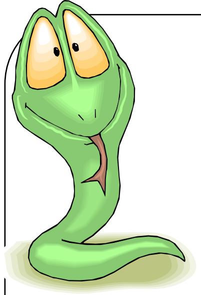
The ‘in’ Operator

- Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- For strings, tests for substrings

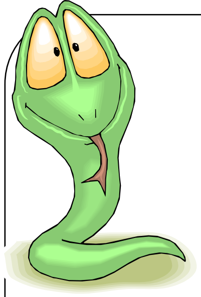
```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```



The + Operator

The + operator produces a *new* tuple, list, or string whose value is the concatenation of its arguments.

```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)

>>> [1, 2, 3] + [4, 5, 6]
 [1, 2, 3, 4, 5, 6]

>>> "Hello" + " " + "World"
 'Hello World'
```

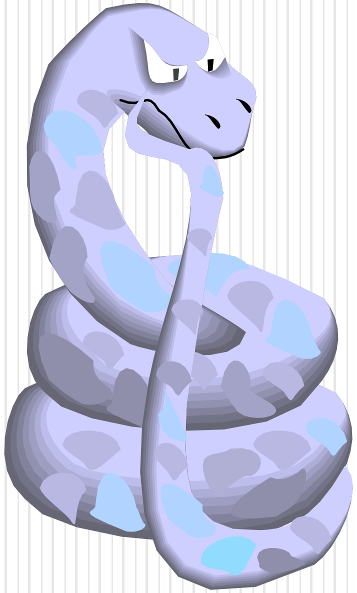


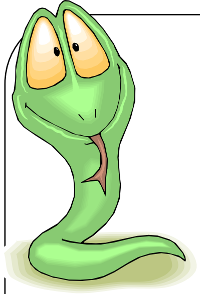
The * Operator

- The * operator produces a *new* tuple, list, or string that "repeats" the original content.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> "Hello" * 3
'HelloHelloHello'
```

Mutability: Tuples vs. Lists

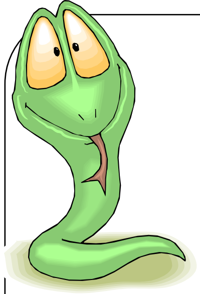




Lists are mutable

```
>>> li = [‘abc’, 23, 4.34, 23]
>>> li[1] = 45
>>> li
 [‘abc’, 45, 4.34, 23]
```

- We can change lists *in place.*
- Name *li* still points to the same memory reference when we’re done.



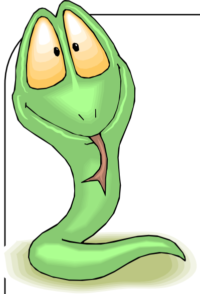
Tuples are immutable

```
>>> t = (23, ‘abc’, 4.56, (2,3), ‘def’)
>>> t[2] = 3.14

Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item assignment
```

- You can’t change a tuple.
- You can make a fresh tuple and assign its reference to a previously used name.

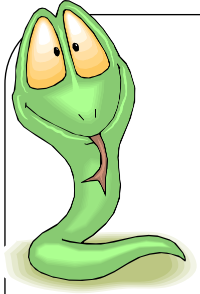
```
>>> t = (23, ‘abc’, 3.14, (2,3), ‘def’)
```

- *The immutability of tuples means they’re faster than lists.*



Operations on Lists Only

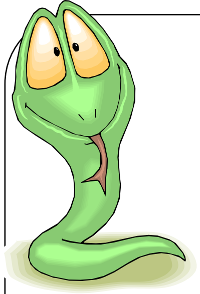
```
>>> li = [1, 11, 3, 4, 5]
>>> li.append(‘a’)
>>> li
[1, 11, 3, 4, 5, ‘a’]

>>> li.insert(2, ‘i’)
>>>li
[1, 11, ‘i’, 3, 4, 5, ‘a’]
```



Operations on Lists Only

Lists have many methods, including index, count, remove, reverse, sort

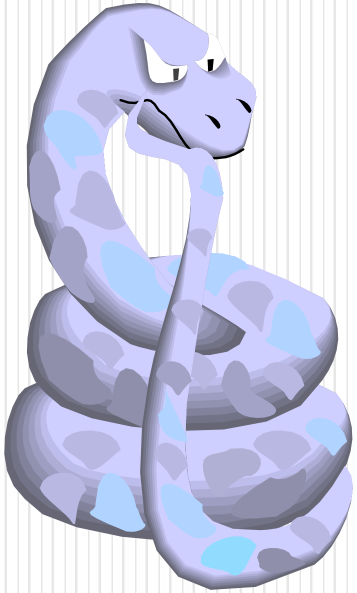
```
>>> li = [‘a’, ‘b’, ‘c’, ‘b’]
>>> li.index(‘b’)  # index of 1st occurrence
1
>>> li.count(‘b’)  # number of occurrences
2
>>> li.remove(‘b’) # remove 1st occurrence
>>> li
  [‘a’, ‘c’, ‘b’]
```

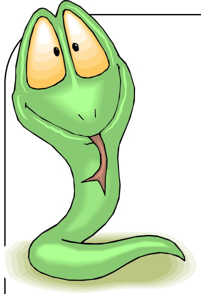


Operations on Lists Only

```
>>> li = [5, 2, 6, 8]

>>> li.reverse()     # reverse the list *in place*
>>> li
  [8, 6, 2, 5]

>>> li.sort()        # sort the list *in place*
>>> li
  [2, 5, 6, 8]

>>> li.sort(some_function)
    # sort in place using user-defined comparison
```

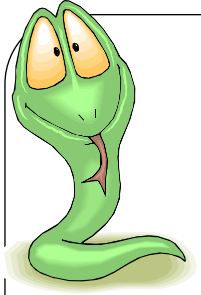
Special Data structure





Dictionaries

- Dictionaries hold key-value pairs
  - Often called maps or hashes. Implemented using hash-tables
  - Keys may be any immutable object, values may be any object
  - Declared using braces
-
  ```
  >>> d={}
  >>> d[0] = "Hi there"
  >>> d["foo"] = 1
  ```

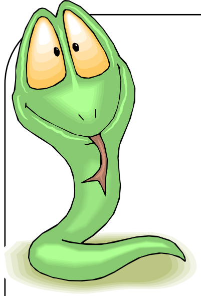


Dictionaries

- Dictionaries (cont.)

-
  ```
  >>> len(d)
  2
  >>> d[0]
  'Hi there'
  >>> d = {0 : "Hi there", 1 :
  "Hello"}
  >>> len(d)
  2
  ```
- Try : Dictionary.py

Functions

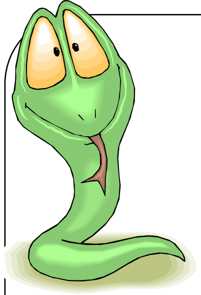


Functions

- Functions are defined with the `def` statement:

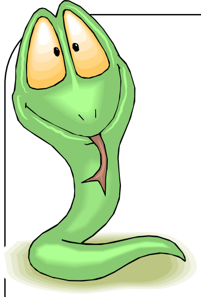
-
  ```
  def foo(bar):
      return bar
  ```

- This defines a function named `foo` that takes a single parameter `bar`

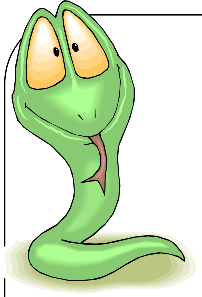


Functions

- A function definition simply places a function object in the namespace
-
```
>>> foo
<function foo at fac680>
>>>
```
- And the function object can obviously be called:
-
```
>>> foo(3)
3
>>>
```



- Try : Sample-P7
  Sample-P8

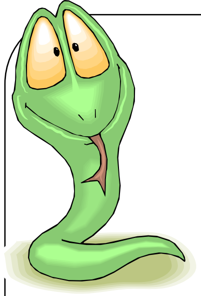


Module

- These are python files containing only function definitions
- Functions can be used in another python files using:
- import modulename
- from modulename import fun-name
- Try: p3.py

  p4.py

File Handling



File Opening

open(“abc.txt”,”r”)

close()

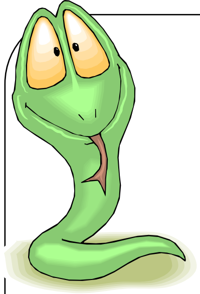
readline()

writeline()

write()

Try:FileHand.py

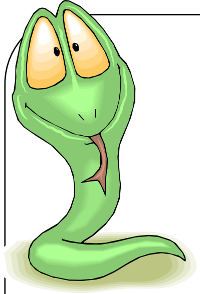
Class handling



Classes

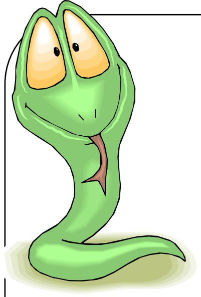
- Classes are defined using the `class` statement
-

```
>>> class Foo:
...     def __init__(self):
...         self.member = 1
...     def GetMember(self):
...         return self.member
...
>>>
```



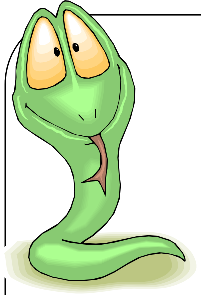
Classes

- A few things are worth pointing out in the previous example:
  - The constructor has a special name `__init__`, while a destructor (not shown) uses `__del__`
  - The `self` parameter is the instance (ie, the `this` in C++). In Python, the self parameter is explicit (c.f. C++, where it is implicit)
  - The name `self` is not required - simply a convention



Classes

- Like functions, a class statement simply adds a class object to the namespace
-
```
>>> Foo
<class __main__.Foo at 1000960>
>>>
```
- Classes are instantiated using call syntax
-
```
>>> f=Foo()
>>> f.GetMember()
1
```

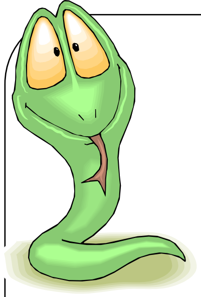


Private and Public Data

- In Python anything with two leading underscores is private
  __a, __my_variable
- Anything with one leading underscore is semiprivate,

_b

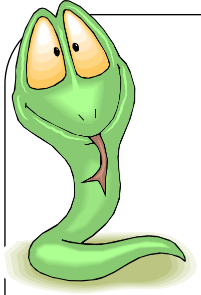
- And no underscores defines it as public



Error Handling

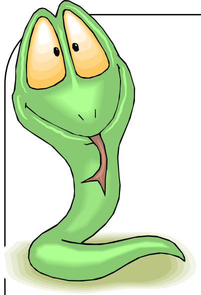
```
try:
        stments
except:
        message
```

Standard libraries



Math

- **import math**
- math.cos(math.pi / 4) 0.70710678118654757
- math.log(1024, 2)
- math.ceil(n)
- max(n1,n2…)
- min(n1,n2…)
- sqrt(n)
- abs(n)



Random

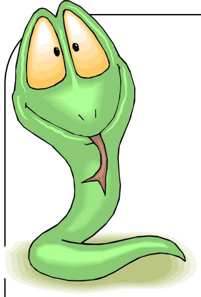
- **import random**
- random.choice(['apple', 'pear', 'banana'])
  - 'apple'
- random.sample(range(100), 10)

# sampling without replacement

  - [30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
- random.random() *# random float*
  - 0.17970987693706186
- random.randrange(6)

# random integer chosen from range(6)

  - 4



OS

- **import os**
- os.getcwd()

# Return the current working directory

- os.chdir('/server/accesslogs')

# Change current working directory

- os.system('mkdir today')
- *# Run the command mkdir in the system shell*

Q&A