Intro to Programming: With Python

(a workshop)

Bibek Gautam

St. Xavier's College, Maitighar

December 10, 2018

Why Python?

Why Python?

```python

from sys import argv
from random import random

HEAD = 1
TAIL = 0

try:
    trials = int(argv[1])
except IndexError:
    print("How many trails do you
    trials = int(input())
results = []
counter = 0

while counter != trials:
    trial = random()
    counter += 1
    if trial < 0.5:
        results.append(TAIL)
    else:
        results.append(HEAD)

```

Figure: Python

```c
#include <stdlib.h>
#include <stdio.h>
#include "llist.h"
#include "tries.h"

llist *createlist(){
    // instantiates a linked list node with NULL vlaues
    llist *newllist = malloc(sizeof(llist));
    if (!newllist){
        fprintf(stderr, "couldn't allocate memory.");
        return NULL;
    }else {
        newllist->this = NULL;
        newllist->next = NULL;
        return newllist;
    }
}
```

Figure: c

Why Python?

Why Python?

Compiled Vs. Interpreted

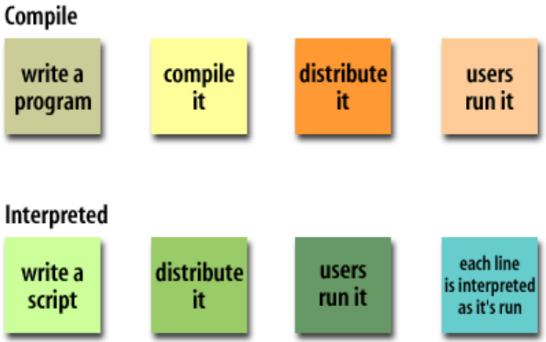

Figure: Compiled Vs. Interpreted

On Linux (Ubuntu/Mint or similar)

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install spyder python3-matplotlib
$ sudo apt-get install  python3-scipy python3-pandas
```

On Windows

- Download and install Anaconda

OR

- Download and install Winpython

Hello world and comments

```
print("Hello World.")
```

← Hello World

```
print("Hi")  # this prints "Hi"
```

← inline

```
''' Everything within this is comment.
This doesn't get evaluated.
It's all comments!
'''
print("hi")
```

← multi-line

Usual Operations

```
+ - * /              # basic arithmetic
//                   # integer division
++ --                # increment
+= -+ *=             #
**                   # power
==  != <= >= < >     # comparision
```

Order of Operation matters!

Variables & Data types

```
age = 5                 # integer
height = 123.2          # float
college = "SXC"         # string
isAlumni = True         # boolean
favNum = 2 + 3j         # complex number

# and a few more
```

Strings

```
name = "John Doe"
sentence = "My name is " + name +"."

print(sentence, end=".")

multiLine = ''' This is a
string with multiple
lines in it '''
```

How would you print : He said "I'm Lucky" ?

```
print("He said \"I'm Lucky\"")
''' \ is a way to escape special meaning (aka
    escape sequence) '''
```

Strings

```
name = "John Doe"
sentence = "My name is " + name +"."

print(sentence, end=".")

multiLine = ''' This is a
string with multiple
lines in it '''
```

How would you print : He said "I'm Lucky" ?

```
print("He said \"I'm Lucky\"")
''' \ is a way to escape special meaning (aka
    escape sequence) '''
```

Strings

```
name = "John Doe"
sentence = "My name is " + name +"."

print(sentence, end=".")

multiLine = ''' This is a
string with multiple
lines in it '''
```

How would you print : He said "I'm Lucky" ?

```
print("He said \"I'm Lucky\"")
''' \ is a way to escape special meaning (aka
    escape sequence) '''
```

Strings

Some useful string methods:

```
.split()
.replace("," , ".")
.find()
.count()
.isalnum()
.int()
.isalpha()
.isdigit()
.strip()
```

Taking inputs:

As simple as it gets...

```
age = input("Enter your age: ")
print("You've lived for about", int(age)*365, "days!")
```

What if i enter characters and not digits?

Taking inputs:

As simple as it gets...

```
age = input("Enter your age: ")
print("You've lived for about", int(age)*365, "days!")
```

What if i enter characters and not digits?

Practice

Here is a problem for you to try:
Get input from user and print it out by splitting it at space character.

Lists

- collection of items
- could be heterogenous
- mutable

```
shopping_list = ['tomatoes', 'potatoes', 'apples', 'juice', 'guava']

print("First item:", shopping_list[0])
```

Lists

List indexing and splicing

```
list[1:3]     # from index 1 to 3 (excluding 3)

list[:3]      # splicing
list[2:]

new_list = list[:]  # makes a copy
list_2 = list       # giving another name
```

Lists

List inside of list

```
matrix = [[1,2,3], [4,5,6], [7,8,9]]

print(matrix[1][0])        # this prints 6 !

matrix.append([10,11,12]) # now matrix is 4*3

matrix.insert(2,[2, 2, 3])  # inserts new row at index 2

a = [1, 4, 5]
matrix = matrix + a          # combining to lists.
```

Lists

Some list useful methods:

```
list.sort()
list.reverse()  # sorting

sorted()    # returns an iterable of sorted items

del(list[4])     # delete item at index 4
max() and min()  # first and last for non numbers)
```

Lists

Some more list useful methods:

```
list = [3,1,4]
len(list)       # length of the list (returns 3)

a = 3 in list   # a is True now
# item in list    evaluates to either true or false.

.find()
.search()
.replace()
```

Lists

string are almost like lists !

Tuples

What are tuples?

- list that cant be changed.
- fixed length, can't be appended or deleted.
- comparable to struct in C
- takes less memory and is faster
- list() and tuple() function to go back and forth between lists and tuple

```
point = (x, y)
student = (name, roll, enrollYr)

# they can be sliced just like lists
student[0]  # gives name of student
point[1]    # gives y co-ordinate of point
```

Tuples

What are tuples?

- list that cant be changed.
- fixed length, can't be appended or deleted.
- comparable to struct in C
- takes less memory and is faster
- list() and tuple() function to go back and forth between lists and tuple

```
point = (x, y)
student = (name, roll, enrollYr)

# they can be sliced just like lists
student[0]  # gives name of student
point[1]    # gives y co-ordinate of point
```

Dictionary

- Dictionary stores key-value pair data
- also known as hash table, look-up table in other languages

```
>>>
>>> capitals = {"Nepal": "Kathmandu",
                "India": "New Delhi",
                "Pakistan": "Islamabad"}

>>> capital["Pakistan"]       # returns Islamabad
>>> capital["France"] = "Rome"
>>> del(capital["France"])    # as you'd expect
```

Dictionary

- Dictionary stores key-value pair data
- also known as hash table, look-up table in other languages

```
>>>
>>> capitals = {"Nepal": "Kathmandu",
                "India": "New Delhi",
                "Pakistan": "Islamabad"}

>>> capital["Pakistan"]       # returns Islamabad
>>> capital["France"] = "Rome"
>>> del(capital["France"])    # as you'd expect
```

Dictionary

- Dictionary stores key-value pair data
- also known as hash table, look-up table in other languages

```
>>>
>>> capitals = {"Nepal": "Kathmandu",
                "India": "New Delhi",
                "Pakistan": "Islamabad"}

>>> capital["Pakistan"]       # returns Islamabad
>>> capital["France"] = "Rome"
>>> del(capital["France"])    # as you'd expect
```

Dictionary

Some dictionary methods:

```
>>> len(dict)
>>> dict.keys()
>>> dict.values()
```

Conditionals - if/else

When your code requires decision making based on conditions

```
>>> if age > 16:
...     print("You're old enough to drive")
... elif age > 25:
...     print("You're old enough to get married")
... else:
...      print("You're not old enough to drive or get married.")
```

Python also has keywords like: and, or, not, True, False

Loops

- loops are for when you have to execute a block of code multiple times.
- there are two modes of loops: **for** and **while** loop.

Loops

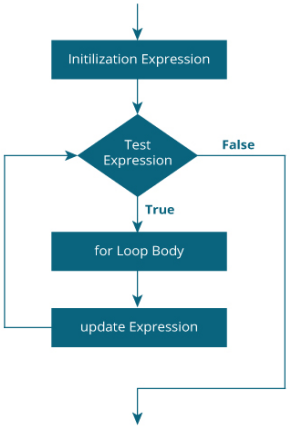

Figure: Basic Logic of Loop

For loop

- For loop iterates through a collection of iterables object or generator function
- for eg. a list is an iterable and so is a string
- dictionary is also an iterable

What are generator functions?

- range() is an example
- another is open() which is used to open files. *We'll come to this later.*

For loop

- For loop iterates through a collection of iterables object or generator function
- for eg. a list is an iterable and so is a string
- dictionary is also an iterable

What are generator functions?

- range() is an example
- another is open() which is used to open files. *We'll come to this later.*

For loop

- For loop iterates through a collection of iterables object or generator function
- for eg. a list is an iterable and so is a string
- dictionary is also an iterable

What are generator functions?

- range() is an example
- another is open() which is used to open files. *We'll come to this later.*

For loop

- For loop iterates through a collection of iterables object or generator function
- for eg. a list is an iterable and so is a string
- dictionary is also an iterable

What are generator functions?

- range() is an example
- another is open() which is used to open files. *We'll come to this later.*

For loop

Let's talk about range()

```
>>> for i in range(4):
...     print(i)
...
0
1
2
3
```

Figure: for loop with range

For loop

Let's talk about range()

```
>>> for i in range(4):
...     print(i)
...
0
1
2
3
```

Figure: for loop with range

range(s, e, stp) with start, end, and step

Loops

Some more range:

```
for i in range(1,11):
    print i
for i in range(1,21,2):
    print i
for i in range(11, 1, -1):  # doesn't include i
    print i
```

Loops

Looping through lists

```
for item in list:
    print(item)
```

But, what if you have multiple dimensional array ??
That's when you need nested loops !

Loops

Looping through lists

```
for item in list:
    print(item)
```

But, what if you have multiple dimensional array ??

That's when you need nested loops !

Loops

Looping through lists

```
for item in list:
    print(item)
```

But, what if you have multiple dimensional array ??

That's when you need nested loops !

Nesting Loops

Nesting means using loops inside of loops :

```
list_of_lists = [ [1,2,3], [4,5,6], [7,8,9] ]
for list in list_of_lists:
    for item in list:
        print(item)

# above prints 1 through 9 when done
```

While loop

When you want to do something
untill a certain condition is true use while loops

- Use while loop when you dont have a tidy data structure to iterate through
- or don't have a relevant generator function.
- when you want to decide when to stop from within the loop
- Let's see what I mean!

While loop

When you want to do something untill a certain condition is true use while loops

- Use while loop when you dont have a tidy data structure to iterate through
- or don't have a relevant generator function.
- when you want to decide when to stop from within the loop
- Let's see what I mean!

While loop

How many perfect cube numbers are less than 100000?

```
this_many = 0       # this counts the number of cube numbers found
cuberoot = 1
while cuberoot**3 < 100000:
    this_many += 1
    cuberoot += 1

print(this_many)    # executes when the loop completes.
```

While loop

How many perfect cube numbers are less than 100000?

```
this_many = 0       # this counts the number of cube numbers found
cuberoot = 1
while cuberoot**3 < 100000:
    this_many += 1
    cuberoot += 1

print(this_many)    # executes when the loop completes.
```

Functions

About functions:

- Functions are similar to what we understand from maths.
- Functions take some input(s) and return some ouput(s)
- input and output could be different type of data

Why use functions?

- Functions allow us to divide the problem into logical chunks of sub-problems that we can separately deal with.
- makes code more readable and more structured!
- makes debugging easy !

Functions

About functions:

- Functions are similar to what we understand from maths.
- Functions take some input(s) and return some ouput(s)
- input and output could be different type of data

Why use functions?

- Functions allow us to divide the problem into logical chunks of sub-problems that we can separately deal with.
- makes code more readable and more structured!
- makes debugging easy !

Functions

About functions:

- Functions are similar to what we understand from maths.
- Functions take some input(s) and return some ouput(s)
- input and output could be different type of data

Why use functions?

- Functions allow us to divide the problem into logical chunks of sub-problems that we can separately deal with.
- makes code more readable and more structured!
- makes debugging easy !

Functions

About functions:

- Functions are similar to what we understand from maths.
- Functions take some input(s) and return some ouput(s)
- input and output could be different type of data

Why use functions?

- Functions allow us to divide the problem into logical chunks of sub-problems that we can separately deal with.
- makes code more readable and more structured!
- makes debugging easy !

Functions

About functions:

- Functions are similar to what we understand from maths.
- Functions take some input(s) and return some ouput(s)
- input and output could be different type of data

Why use functions?

- Functions allow us to divide the problem into logical chunks of sub-problems that we can separately deal with.
- makes code more readable and more structured!
- makes debugging easy !

Functions

About functions:

- Functions are similar to what we understand from maths.
- Functions take some input(s) and return some ouput(s)
- input and output could be different type of data

Why use functions?

- Functions allow us to divide the problem into logical chunks of sub-problems that we can separately deal with.
- makes code more readable and more structured!
- makes debugging easy !

Functions

About functions:

- Functions are similar to what we understand from maths.
- Functions take some input(s) and return some ouput(s)
- input and output could be different type of data

Why use functions?

- Functions allow us to divide the problem into logical chunks of sub-problems that we can separately deal with.
- makes code more readable and more structured!
- makes debugging easy !

Functions

Let's see an example

```
def toPercent(num, denum):
    ''' Takes a fraction and returns
    equivalent percentage '''
    percent = num/denum  * 100
    percent = int(percent)
    return percent      # return is used to output

print("4/5 = ", toPercent(4,5), "%.")
```

Namescope

- Namescope is like a dictionary that maps variable names to their values
- Every function has its own namespace.
- which means variable declared in one function can't be accessed from another.

Overview OOP

Figure: OOP in Games

Objects heirarchy

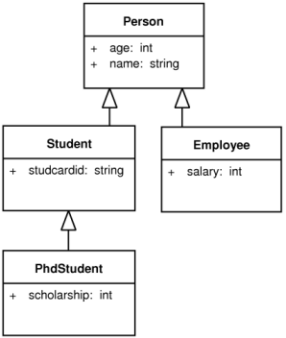

Figure: Objects and inheritence

Objects heirarchy

Attributes and methods
Variables *encapculated* inside a class are its attributes. And functions *encapculated* within a class are called its methods.

Parent Class
The class above a class in class heirarchy is called its parent.

Child class
The classes that derive from a class are its children.

Inheritence
Child class retains all the methods and attributes from the parents. This is known as inheritence.

To be continued...

Thank You!