18ES611 Embedded System Programming

Sarath tv

Why Python?

Readability and ease-of-maintenance

- Python focuses on well-structured easy to read code
- Easier to understand source code…
- ..Hence easier to maintain code base
- Portability
- Scripting language hence easily portable
- Python interpreter is supported on most modern os's
- Extensibility with libraries
- Large base of third-party libraries that greatly extend

Functionality. Eg., Numpy, scipy etc.

Introduction to python

Python allows you to use variables without declaring them (i.e., it determines types implicitly), and it relies on indentation as a control structure.

Python Interpreter

The system component of Python is the interpreter.

The interpreter is independent of your code and is required to execute your code.

Interpreters and compilers are similar, since they both recognize and process source code.

However, a compiler does not execute the code like and interpreter does. Instead, a compiler simply converts the source code into machine code, which can be run directly by the operating system as an executable program. **Interpreters bypass the compilation process and execute the code directly.**

Since interpreters read and execute code in a single step, they are useful for running scripts and other small programs

Script vs. command line

Code can be written in a python script that is interpreted as a block.

- Code can also be entered into the Python command line interface.
- You can exit the command line with Ctrl-z on windows and Ctrl-d on unix
- For complex projects use an IDE (For example, PyCharm, Jupyter notebook).
- PyCharm is great for single-developer projects
- Jupyter is great sharing code and output with markup

Variables and Objects

Variables are the basic unit of storage for a program.

- Variables can be created and destroyed.
- At a hardware level, a variable is a reference to a location in memory.
- Programs perform operations on variables and alter or fill in their values.
- Objects are higher level constructs that include one or more variables and the set of operations that work on these variables.
- An object can therefore be considered a more complex variable.

Classes vs. Objects

Every Object belongs to a certain class.

Classes are abstract descriptions of the structure and functions of an object.

Objects are created when an instance of the

class is created by the program.

- For example, “Fruit” is a class while an “Apple” is an object.

Mutable vs. Immutable

immutable i.,e cannot be directly changed

Mutable can be changed in place.

```
>>> S[0]
'H'
>>> S[0] = 'h'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> L[1]
3.1400000000000001
>>> L[1] = 3.145
```

Core data types

Numbers

Strings

Lists

Dictionaries

Tuples

Files

Numbers

Can be integers, decimals (fixed precision), floating points (variable precision), complex numbers etc.

Simple assignment creates an object of number type such as:

a = 3

b = 4.56

Supports simple to complex arithmetic operators.

Assignment via numeric operator also creates a number object:

c = a / b

a, b and c are numeric objects.

Try dir(a) and dir(b) . This command lists the functions available for these objects.

Strings

A string object is a ‘sequence’, i.e., it’s a list of items where each item has a defined position.

Each character in the string can be referred, retrieved and modified by using its position.

This order id called the ‘index’ and always starts with 0.

```
>>> S = 'Hello'
>>> len(S)
5
>>> S[0]
'H'
>>> S[4]
'o'
>>> S[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

```
>>> S[-1]
'o'
>>> S[3:]
'lo'
>>> S[2:5]
'llo'
```

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

String objects support concatenation and repetition operations.

```
>>> S + 'World!'
'HelloWorld!'
>>> S + ' World!'
'Hello World!'
>>> S * 4
'HelloHelloHelloHello'
>>> S + ' World! ' * 4
'Hello World!  World!  World!  World! '
>>> (S + ' World! ') * 4
'Hello World! Hello World! Hello World! Hello World! '
```

Lists

List is a more general sequence object that allows the individual items to be of different types.

Lists have no fixed size and can be expanded or contracted as needed.

Items in list can be retrieved using the index.

Lists can be nested just like arrays, i.e., you can have a list of lists

Define lists are using square brackets and commas

- Simple list:

```
>>> L = [123, 3.14, 'Hello']
>>> L
[123, 3.1400000000000001, 'Hello']
```

```
>>> L[0]
123
```

- Nested list:

```
>>> DDL = [[1,2,3],
... [4,5,6],
... [7,8,9]]
>>> DDL
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
>>> DDL[2][1]
8
```

Dictionaries

Dictionaries are unordered mappings of **'Name : Value**' associations.

Comparable to hashes and associative arrays in other languages.

Intended to approximate how humans remember associations.

```
>>> D = {'name':'apple','color':'red','taste':'sweet','number':'5'}
>>> D['name']
'apple'
>>> D
{'color': 'red', 'taste': 'sweet', 'name': 'apple', 'number': '5'}
```

Tuples

Tuples are immutable lists.

Maintain integrity of data during program execution.

Define tuples using parentheses and commas

Tuple • An *immutable* ordered sequence of items . Items can be of mixed types, including collection types

Strings • An *immutable* ordered sequence of chars .

List • A *Mutable* ordered sequence of items of mixed types

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’
```

>>> 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 a copy of subset

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

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

>>> t[1:4]

(‘abc’, 4.56, (2,3))

- You can also use negative indices

>>> t[1:-1]

(‘abc’, 4.56, (2,3))

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

Omit 1st index to make a copy starting from the beginning of container

```
>>> t[:2]
(23, ‘abc’)
```

Omit 2nd index to make a copy starting at 1st index and going to end of the container

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

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

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

- Note the difference between these two lines for mutable sequences

```
>>> l2 = l1 # Both refer to same ref,
# changing one affects both
>>> l2 = l1[:] # Independent copies, two refs
```

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

List 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 -topleveltu[
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 in List only

```
>>> li = [1, 11, 3, 4, 5]
>>> li.append('a') # Note the method syntax
>>> li
[1, 11, 3, 4, 5, 'a']
>>> li.insert(2, 'i')
>>>li
[1, 11, 'i', 3, 4, 5, 'a']
```

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’]
```

```
>>> 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]
```

Functions in python

Lambda

Sometimes you need a simply arithmetic function

- Its silly to write a method for it, but redundant not too
- With lambda we can create quick simple functions
- Facts

– Lambda functions can only be comprised of a single expression

– No loops, no calling other methods

– Lambda functions can take any number of variables

Syntax:

lambda **param1**,…,**paramn** : **expression**

```
#Example 1
square_func = lambda x : x**2
square_func(4)                          #return: 16

#Example 2
close_enough = lambda x, y : abs(x - y) < 3
close_enough(2, 4)                      #return: True

#Example 3
def get_func(n) :
    return lambda x : x * n + x % n
my_func = get_func(13)
my_func(4)                              #return: 56
```

Defining functions

def myfunction():

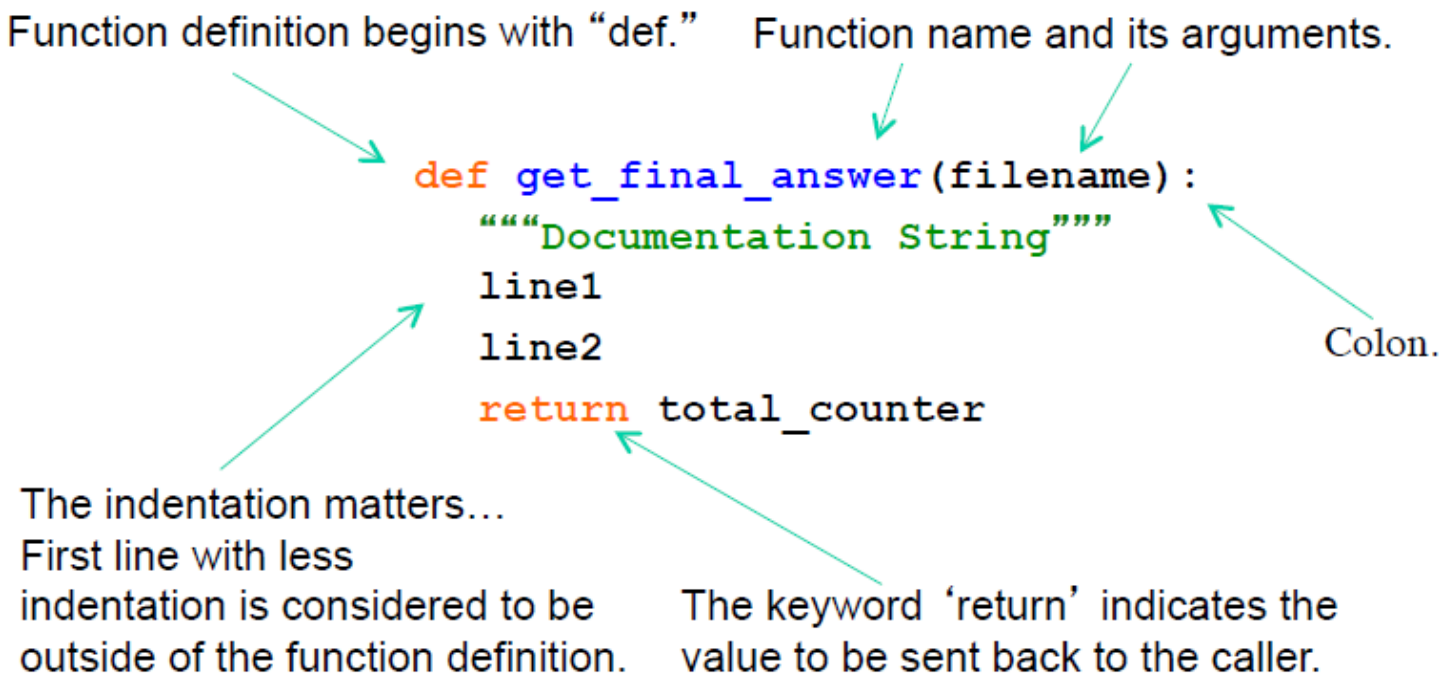
print (“Printed from inside a fun

call the function

myfunction()

Incase of arguments- it can be o

sequence also



No header file or declaration of types of function or arguments

```
def printinfo( name, age = 35 ):
"This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;
# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

Control statements

The range function

range(x) := [0,1,2,…,x---1]

The Continue statement

Continue := continue with the next iteration

```
for I in range(5):
    if I == 3:
     continue
    print (I)
```

The break statement

break := break out of the loop

```
for I in range(5):
    if I > 2:
        break
    print(I)
```

If statement

```
Temp = 5
If temp < 10:
    Print "It is cold!"
```

if...else statement

```
Temp = 20
If temp < 10:
    Print "It is cold!"
else:
    Print "It feels good!“
```

if...elif...else statement

```
Temp = 40
If temp < 10:
    print"It is cold!"
Elif temp > 30:
    Print "It is hot!“
else:
    Print "It feels good!“
```

For loops

```
For I in [0,1,2,3,4]:
    Print i
```

While loops

```
I = 0
While (I < 5):
    I = I + 1
    Print i
```

_____

While loops with break and continue

```
I = 0
while (I < 5):
    I = I + 1
    If I == 3:
        continue
    If I > 4:
        break
        Print i
```

Modules in python

if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. **-a *script*.**

As program gets longer, split it into several files for easier maintenance.

You may also want to use a handy function that you've written in several programs without copying its definition into each program.

Python has a way to put **definitions in a file and use them in a script** or in an interactive instance of the interpreter.

Such a **file is called a *module*;** *definitions from a module can be imported into other modules or into the main module* (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

modules

A module allows you to logically organize your Python code.

Grouping related code into a module makes the code easier to understand and use.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

The Python code for a module named ***name*** normally resides in a file named ***name.py***.

The import Statement

You can use any Python source file as a module by executing an import statement in some other Python source file. The import has the following syntax −

```
import module1
```

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module.

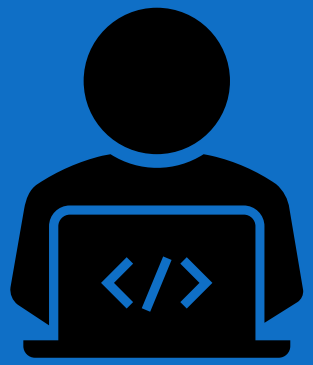
The *from...import* Statement

Python's from statement lets you import specific attributes from a module into the current namespace. The from...import has the following syntax –

from modname import name1

import ABC as XXX

fib = fibo.fib -local name



THANK YOU!!!!!