Introduction to Python Programming

Data Structures & Algorithms

Lecture 1

Topics to Cover

- Revisiting Basics
  - Python Objects - identifiers
  - Sequence types - Lists, Tuple, Sets, Strings
  - Dictionary Class
  - Operators for Sequences & Dictionaries
  - Polymorphic Functions
  - File Read & Write
- Advanced
  - Exception Handling
  - Iterators & Generators
  - Python Conveniences
  - Scopes & Namespaces

Introduction

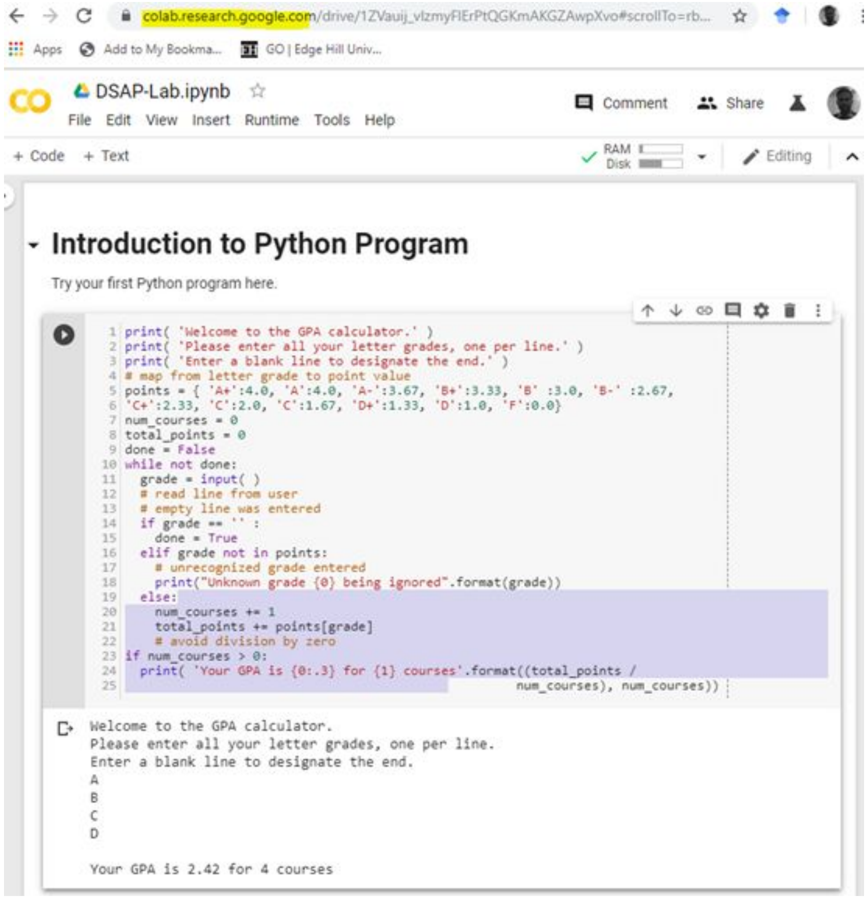
- What is Python?

  It is an *interpreted* programming language designed for high readability and faster prototyping leading to production-ready software development.

- Why Python?
  - Works on multiple platforms - Windows, Linux, Mac etc.
  - Simple syntax similar to English language
  - Write programs with fewer lines compared to other programming language
  - Runs on interpreter - code can be executed as soon as it is written.
- Good to Know
  - Most recent version is Python 3 with some major updates over Python 2.
  - IDES - pyCharm, Eclipse, Jupyter

How and Where to Write Programs?

- Codes could be executed online on Google Colab. Google Login will be required.
- Create a New Python 3 Notebook: File -> New Python 3 Notebook
- An Example Program Sheet is provided for reference here.
- We will use Jupyter Notebook for writing programs.
- Go through the Google Colab tutorial to understand more about its use.
- You can also write your codes on other Python IDEs as well.



Python Introduction

- An *interpreted* programming language
- # is used for inserting comments
- “”” … “”” or ‘’’ … ‘’’ can be used multi-line comments.
- ‘\’ can be used for continuing command on multiple lines
- A code block is tab indented within a control structure
- An object oriented language and ***classes*** form the basis for all data types.

Identifiers

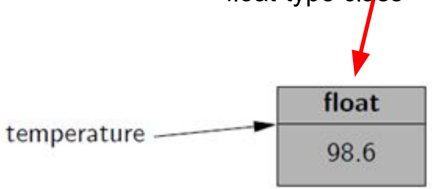
- ***Identifiers*** are variable names - case sensitive, combination of letters,numbers and underscores.
- Python identifier is most similar to a ***reference variable*** in Java or a ***pointer variable*** in C++.
  - Each identifier is implicitly associated with the ***memory address*** of the object to which it refers.
- Python is ***dynamically typed*** language as there is no advance declaration associating an identifier with a particular data type.

Reserved Words								
False	as	continue	else	from	in	not	return	yield
None	assert	def	except	global	is	or	try	
True	break	del	finally	if	lambda	pass	while	
and	class	elif	for	import	nonlocal	raise	with	

Table 1.1: A listing of the reserved words in Python. These names cannot be used as identifiers.

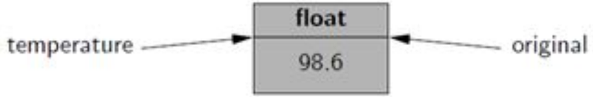
Creating and Using Objects

98.6 instantiates a float-type class



- Create a new instance of a class - ***instantiation***
- Python ***literals*** for designating new instances.

```
> temperature = 98.6
```



- Alias: assigning a second identifier to an existing object

```
> original = temperature
```

- Python supports various ways of calling methods:
  - sort(data) # standard call to methods / functions
  - data.sort() # member functions

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

Commonly used built-in classes for Python

Types of Methods

- ***Accessors*** - return information about the state of an object, but do not change the state
- ***Mutators*** or update methods: change the state of the object.
- A class is ***immutable*** if each object of that class has a fixed value upon instantiation that cannot subsequently be changed. Example: float, int, bool, str class

```python
class Student():
  def __init__(self, name=None, age=None, no_sub=None):
    self._name = name
    self._age = age
    self._no_subj = no_sub

  def get_attrib(self):    # Accessor
    attrib = [self._name, self._age, self._no_subj]
    return attrib
  def set_attrib(self,name,age, no_sub):  # Mutator
    self._name = name
    self._age = age
    self._no_subj = no_sub
###############

stud1 = Student("Tom", 12, 4)
stud2 = Student("Harry", 10, 2)
print(stud1.get_attrib())
print(stud2.get_attrib())
stud1.set_attrib("Tom.H", "13", 5)
stud2.set_attrib("Harry.Mitchel", "11", 3)
print(stud1.get_attrib())
print(stud2.get_attrib())
```

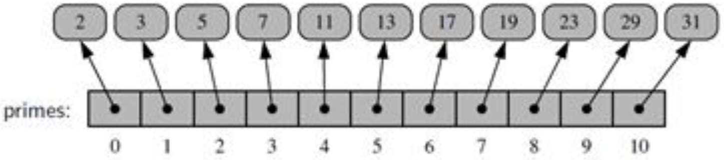
```
['Tom', 12, 4]
['Harry', 10, 2]
['Tom.H', '13', 5]
['Harry.Mitchel', '11', 3]
```

Sequence Type Classes

- The `list` class
- The `tuple` class
- The `str` class
- The `set` / `frozenset` class

The `list` class

- A list instance stores a sequence of objects.
- A list is a ***referential structure -*** as it technically stores a sequence of references to its elements.
- Lists are ***array-based sequences*** and are ***zero-indexed***, thus a list of length n has elements indexed from 0 to n−1 inclusive.
- Python uses the characters [ ] as delimiters for a list literal, with [ ] itself being an empty list.
- The `list()` constructor produces an empty list by default. However, the constructor will accept any parameter that is of an iterable type.
- The list is a ***mutable*** sequence - The elements can be added or removed from the sequence.



> prime = [2,3,5,7,11,13,17,19,23,29,31]

```python
d = list('hello')
print(d)
a = ['red', 'green', 'blue']
print(a)
b = [10,12,15]
c = [a,b]
print(c)
g = ["Tom", 32, 5, "london"]
print(g)
```

```
['h', 'e', 'l', 'l', 'o']
['red', 'green', 'blue']
[['red', 'green', 'blue'], [10, 12, 15]]
['Tom', 32, 5, 'london']
```

```python
# Lists are mutable
a = [2,3,4,5]
a.append(6)
print(a)
a.remove(3)
print(a)
a.insert(1,10)
print(a)


```

```
[2, 3, 4, 5, 6]
[2, 4, 5, 6]
[2, 10, 4, 5, 6]
```

The `tuple` class

- An ***immutable*** version of a sequence. It can not be changed. Elements can not be added to or removed from the sequence.

- parentheses () delimit a tuple, with () being an empty tuple.

- To express a tuple of length one as a literal, a comma must be placed after the element, but within the parentheses.
  For example, (17,) is a one-element tuple.

```python
# Tuples
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5, 6, 7 )
tup3 = "a", "b", "c", "d"
print ("tup1[0]: ", tup1[0])
print ("tup2[1:5]: ", tup2[1:5])
print(tup3)
tup4 = (17)
print(type(tup4))
tup5 = (17,)
print(type(tup5))

```

```
tup1[0]:  physics
tup2[1:5]:  (2, 3, 4, 5)
('a', 'b', 'c', 'd')
<class 'int'>
<class 'tuple'>
```

```python
# Tuple are immutable - they can not be changed.
a = (2,3,4,5)
a.append(5) # Gives error
print(a)

b = list(a)
b.append(5)
c = tuple(b)
print(c)


```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-31-267031d8f0aa> in <module>()
      1 a = (2,3,4,5)
----> 2 a.append(5) # Gives error
      3 print(a)
      4
      5 b = list(a)

AttributeError: 'tuple' object has no attribute 'append'
```

The `str` class

- represents an immutable sequence of characters.
- Examples: “hello”, ‘hello’, “Don’t Worry”, ‘Don\’t worry’, ‘C:\\Python\\’ (notice the **escape characters**)
- Unicode characters can also be included. E.g. ‘20\u20AC’ for string ‘20€’
- ‘ ’ ’ and “ ” ” for multi-line comments
- Individual characters of the string can be accessed using [] operator.

```python
# Strings

firstName = 'john'
lastName = "smith"
message = """This is a string that will span across
 multiple lines. Using newline characters
and no spaces for the next lines. The end
 of lines within this string also count as a
 newline when printed"""

print(firstName)
print(lastName)
print(message)


var1 = 'Hello World!'
var2 = 'RhinoPython'

print(var1[0])
print(var2[1:5])

c = 'Don\'t Worry'
print(c)

d = "C:\\Python\\Directory\\"
print(d)
```

```
john
smith
This is a string that will span across
 multiple lines. Using newline characters
and no spaces for the next lines. The end
 of lines within this string also count as a
 newline when printed
H
hino
Don't Worry
C:\Python\Directory\
```

The `set` and `frozenset` classes

- The `set` class represents the mathematical notion of a set, namely a collection of elements, without duplicates, and without an inherent order to those elements.

- The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set. It is based on a data structure called "***Hash Tables***".

- It has two restrictions:
  - It does not maintain the elements in any particular order.
  - Only instances of immutable types can be added into a python set: int, floats, strings, tuples and frozensets.

- The **frozenset** class is an immutable form of the set type.
- Sets use curly braces { and } as delimiters for a set.
- Examples:
  - {17}, {'red', 'green', 'blue'}
  - {} does not represent an empty set for historical reasons. It represents an empty dictionary.
  - set() produces an empty set. set('hello') produces {'h', 'e', 'l', 'l', 'o'}.

```python
a = {1,2,3}
b = {4,5}
c = {1.0, "Hello", (1,2,3)} # tuple is immutable
print(c)

d = set([1,2,3,4])
print(d)
print(set('hello'))

e = {}
print(type(e))

f = set() # create an empty set
print(type(f))

f.add(4)  # modify set
f.update([5,6,7])
print(f)

g = frozenset([1,2,3,4])
print(g)
print(type(g))

h = {1.0, "hello", g} # g is immutable
print(h)


my_set = {1,2, [3,4]}  # gives an error
```

```
{1.0, (1, 2, 3), 'Hello'}
{1, 2, 3, 4}
{'l', 'h', 'e', 'o'}
<class 'dict'>
<class 'set'>
{4, 5, 6, 7}
frozenset({1, 2, 3, 4})
<class 'frozenset'>
{1.0, frozenset({1, 2, 3, 4}), 'hello'}
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-58-1da8b0d1f8d4> in <module>()
     26
     27
---> 28 my_set = {1,2, [3,4]}  # gives an error

TypeError: unhashable type: 'list'
```

The `dict` class

- Represents a ***dictionary*** or a ***mapping,*** from a set of distinct keys to associated values.
- A dictionary literal also uses curly braces.

```
> a = {‘ga’:’Irish’, ‘de’:’German’}
```

- The literal form { } produces an empty dictionary.

- The constructor for the dict class accepts an existing mapping as a parameter.

- Alternatively, the constructor accepts a sequence of key-value pairs as a parameter, as in dict(pairs) with pairs:

```
> b = [( ‘ga’ , ‘Irish’ ), ( ‘de’ , ‘German’ )]
```

```python
# Python Dictionaries

numbers = dict(x=5, y=0)
print('numbers =', numbers)
print(type(numbers))

empty = dict()
print('empty =', empty)
print(type(empty))

# Another way to create dictionaries
n2 = {'y': 0, 'x': 5}
print(n2)
print(type(n2))
empty = {}  # empty dictionaries
print(type(empty))

# create dictionary using iterables
numbers1 = dict([('x', 5), ('y', -5)])
print('numbers1 =',numbers1)

# keyword argument is also passed
numbers2 = dict([('x', 5), ('y', -5)], z=8)
print('numbers2 =',numbers2)

# zip() creates an iterable in Python 3
numbers3 = dict(dict(zip(['x', 'y', 'z'], [1, 2, 3])))
print('numbers3 =',numbers3)

# Create dictionaries using mappings
numbers1 = dict({'x': 4, 'y': 5})
print('numbers1 =',numbers1)

# you don't need to use dict() in above code
numbers2 = {'x': 4, 'y': 5}
print('numbers2 =',numbers2)

# keyword argument is also passed
numbers3 = dict({'x': 4, 'y': 5}, z=8)
print('numbers3 =',numbers3)


#using dict()
my_dict = dict({1:'apple', 2:'ball'})
print(my_dict)
```

Output

```
numbers = {'x': 5, 'y': 0}
<class 'dict'>
empty = {}
<class 'dict'>
{'y': 0, 'x': 5}
<class 'dict'>
<class 'dict'>
numbers1 = {'x': 5, 'y': -5}
numbers2 = {'x': 5, 'y': -5, 'z': 8}
numbers3 = {'x': 1, 'y': 2, 'z': 3}
numbers1 = {'x': 4, 'y': 5}
numbers2 = {'x': 4, 'y': 5}
numbers3 = {'x': 4, 'y': 5, 'z': 8}
{1: 'apple', 2: 'ball'}
Jack
26
```

Expressions, Operators & Precedence

- Logical Operator: not, and, or
- Equality Operators: is, is not, ==, !=
- Comparison operators: <, <=, >, >=
- Arithmetic Operators: +, - , *, /, //, %
- Bitwise Operators: ~,&, |, ^, <<, >>

Sequence Operators

- Python relies on ***zero-indexing*** of sequences, thus a sequence of length n has elements indexed from 0 to $n - 1$ inclusive.

- Python also supports the use of negative indices, which denote a distance from the end of the sequence; index -1 denotes the last element, index -2 the second to last, and so on.

- **Slicing:** half-open intervals. Example: data[3:8] denotes a subsequence including the five indices: 3,4,5,6,7.

- All sequences define comparison operations based on lexicographic order, performing an element by element comparison until the first difference is found. For example, [5, 6, 9] < [5, 7] because of the entries at index 1

s[j]	element at index j
s[start:stop]	slice including indices [start,stop)
s[start:stop:step]	slice including indices start, start + step, start + 2*step, ..., up to but not equalling or stop
s + t	concatenation of sequences
k * s	shorthand for s + s + s + ... (k times)
val **in** s	containment check
val **not in** s	non-containment check

s == t	equivalent (element by element)
s != t	not equivalent
s < t	lexicographically less than
s <= t	lexicographically less than or equal to
s > t	lexicographically greater than
s >= t	lexicographically greater than or equal to

Operators for Sets

key **in** s	containment check
key **not in** s	non-containment check
s1 == s2	s1 is equivalent to s2
s1 != s2	s1 is not equivalent to s2
s1 <= s2	s1 is subset of s2
s1 < s2	s1 is proper subset of s2
s1 >= s2	s1 is superset of s2
s1 > s2	s1 is proper superset of s2
s1 | s2	the union of s1 and s2
s1 & s2	the intersection of s1 and s2
s1 − s2	the set of elements in s1 but not s2
s1 ˆ s2	the set of elements in precisely one of s1 or s2

Operators for Dictionaries

d[key]	value associated with given key
d[key] = value	set (or reset) the value associated with given key
del d[key]	remove key and its associated value from dictionary
key **in** d	containment check
key **not in** d	non-containment check
d1 == d2	d1 is equivalent to d2
d1 != d2	d1 is not equivalent to d2

Extended Assignment Operators

```
alpha = [1, 2, 3]
beta = alpha                # an alias for alpha
beta += [4, 5]              # extends the original list with two more elements
beta = beta + [6, 7]        # reassigns beta to a new list [1, 2, 3, 4, 5, 6, 7]
print(alpha)                # will be [1, 2, 3, 4, 5]
```

Operator Precedence

- Chained assignment
  - x = y = 0
  - 1 <= x + y <= 10

Operator Precedence		
	Type	**Symbols**
1	member access	expr.member
2	function/method calls container subscripts/slices	expr(...) expr[...]
3	exponentiation	**
4	unary operators	+expr, −expr, ˜expr
5	multiplication, division	*, /, //, %
6	addition, subtraction	+, −
7	bitwise shifting	<<, >>
8	bitwise-and	&
9	bitwise-xor	^
10	bitwise-or	|
11	comparisons containment	**is**, **is not**, ==, !=, <, <=, >, >= **in**, **not in**
12	logical-not	**not** expr
13	logical-and	**and**
14	logical-or	**or**
15	conditional	val1 **if** cond **else** val2
16	assignments	=, +=, −=, *=, etc.

Control Flow: If, While, For, Break & Continue

```
if first_condition:
    first_body
elif second_condition:
    second_body
elif third_condition:
    third_body
else:
    fourth_body
```

```
while condition:
    body
```

```
for element in iterable:
    body
```

- A **break** statement terminates a most immediately closing while or for loop.
- A **continue** statement causes the current iteration of a loop body to stop, but with subsequent passes of loop proceeding as expected.

Functions

- The term ***function*** is used to describe a traditional, stateless function that is invoked without the context of a particular class or an instance of that class, such as:

```
> b =  sorted(data)
```

- The term ***method*** is used to describe a member function that is invoked upon a specific object using an object-oriented message passing syntax, such as:

```
> c = data.sort()
```

- The identifiers used to describe the expected parameters are known as ***formal parameters.***

- The objects sent by the caller when invoking the function are the ***actual parameters.***

- ***Polymorphic functions***: more than one possible calling signatures - default parameter values.

Default Parameter:

```
def foo(a, b=15, c=27):
```

Valid calls:

```
foo(4,12,8)
foo(4)
```

```
> def foo (a=10, b=20, c=30):
```

Positional Argument:

`foo(5)` will execute `foo(a=5,b=20,c=30)`

Keyword Argument:

`foo(c=5)` will execute `foo(a=10,b=20,c=5)`

```
def bar(a, b=5, c)
```

Illegal function signature !!

Signature of the function

```
def count(data, target):
  n = 0
  for item in data:
    if item == target:
      n += 1
  return n
```

Body of the function

End of the function

Input / Output, File I/O

- Input from Keyboard: input()
- Output to Console: print()

- File Handling
  - File is a named location on the disk which is used to store data permanently.
  - Following operations can be performed on a file:
    - Open a file
    - Read a file
    - Write a file
    - Close a file

```python
# Input / Output
print("Hello World!", "first time here?")
x = input("something:")
print(x)
print(type(x))

year = 2016
event = 'Referendum'
print(f'Results of the {year} {event}')


yes_votes = 42_572_654
no_votes = 43_132_495
percentage = yes_votes / (yes_votes + no_votes)
print('{:-9} YES votes  {:2.2%}'.format(yes_votes, percentage))
```

Example 1

```
Hello World! first time here?
something:10
10
<class 'str'>
Results of the 2016 Referendum
 42572654 YES votes  49.67%
```

```python
# File input/output examples

f = open("test.txt", 'w')
f.write("Hello Python!\n")
f.write("Today is Monday\n")
f.write("Hello World!\n")
f.close()

f = open("test.txt", 'r')
print(f.read())
f.close()
```

```
Hello Python!
Today is Monday
Hello World!
```

Example 2

Common Built-In Functions	
Calling Syntax	**Description**
abs(x)	Return the absolute value of a number.
all(iterable)	Return True if bool(e) is True for each element e.
any(iterable)	Return True if bool(e) is True for at least one element e.
chr(integer)	Return a one-character string with the given Unicode code point.
divmod(x, y)	Return (x // y, x % y) as tuple, if x and y are integers.
hash(obj)	Return an integer hash value for the object (see Chapter 10).
id(obj)	Return the unique integer serving as an "identity" for the object.
input(prompt)	Return a string from standard input; the prompt is optional.
isinstance(obj, cls)	Determine if obj is an instance of the class (or a subclass).
iter(iterable)	Return a new iterator object for the parameter (see Section 1.8).
len(iterable)	Return the number of elements in the given iteration.
map(f, iter1, iter2, ...)	Return an iterator yielding the result of function calls f(e1, e2, ...) for respective elements $e1 \in iter1, e2 \in iter2, \ldots$
max(iterable)	Return the largest element of the given iteration.
max(a, b, c, ...)	Return the largest of the arguments.
min(iterable)	Return the smallest element of the given iteration.
min(a, b, c, ...)	Return the smallest of the arguments.
next(iterator)	Return the next element reported by the iterator (see Section 1.8).
open(filename, mode)	Open a file with the given name and access mode.
ord(char)	Return the Unicode code point of the given character.
pow(x, y)	Return the value x^y (as an integer if x and y are integers); equivalent to x ** y.
pow(x, y, z)	Return the value (x^y mod z) as an integer.
print(obj1, obj2, ...)	Print the arguments, with separating spaces and trailing newline.
range(stop)	Construct an iteration of values 0, 1, ..., stop − 1.
range(start, stop)	Construct an iteration of values start, start + 1, ..., stop − 1.
range(start, stop, step)	Construct an iteration of values start, start + step, start + 2*step, ...
reversed(sequence)	Return an iteration of the sequence in reverse.
round(x)	Return the nearest int value (a tie is broken toward the even value).
round(x, k)	Return the value rounded to the nearest 10^{-k} (return-type matches x).
sorted(iterable)	Return a list containing elements of the iterable in sorted order.
sum(iterable)	Return the sum of the elements in the iterable (must be numeric).
type(obj)	Return the class to which the instance obj belongs.

Built-in File I/O commands

Calling Syntax	Description
fp.read()	Return the (remaining) contents of a readable file as a string.
fp.read(k)	Return the next *k* bytes of a readable file as a string.
fp.readline()	Return (remainder of) the current line of a readable file as a string.
fp.readlines()	Return all (remaining) lines of a readable file as a list of strings.
for line **in** fp:	Iterate all (remaining) lines of a readable file.
fp.seek(k)	Change the current position to be at the k^{th} byte of the file.
fp.tell()	Return the current position, measured as byte-offset from the start.
fp.write(string)	Write given string at current position of the writable file.
fp.writelines(seq)	Write each of the strings of the given sequence at the current position of the writable file. This command does *not* insert any newlines, beyond those that are embedded in the strings.
print(..., file=fp)	Redirect output of print function to the file.

Exception Handling

- An exception is an event which occurs during the execution of a program that disrupts its normal flow.

- Exception Handling is the process of dealing with run-time errors and faults gracefully.

- It includes two major tasks:
  - Throw suitable Exceptions – `raise(), assert()`
  - Execute suitable functions by catching these exceptions: `try … except` block

Raise an Exception

```python

# Exception Handling

import math

def my_sqrt(x):
  if not isinstance(x, (int, float)):
    raise TypeError('x must be numeric')
  elif x < 0:
    raise ValueError('x can not be negative')

  return math.sqrt(x)

###############


x = 25
print('sqrt of {} = {}'.format(x, my_sqrt(x)))

x = -25;
print('sqrt of {} = {}'.format(x, my_sqrt(x)))

x = "twenty"
print('sqrt of {} = {}'.format(x, my_sqrt(x)))

```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-37-fe2a2904dbc2> in <module>()
     17
     18 x = -25;
---> 19 print('sqrt of {} = {}'.format(x, my_sqrt(x)))
     20
     21 #x = "twenty"

<ipython-input-37-fe2a2904dbc2> in my_sqrt(x)
      6     raise TypeError('x must be numeric')
      7   elif x < 0:
----> 8     raise ValueError('x can not be negative')
      9
     10   return math.sqrt(x)

ValueError: x can not be negative
```

```
sqrt of 25 = 5.0
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-36-b996e4c75f60> in <module>()
     20
     21 x = "twenty"
---> 22 print('sqrt of {} = {}'.format(x, my_sqrt(x)))

<ipython-input-36-b996e4c75f60> in my_sqrt(x)
      4 def my_sqrt(x):
      5   if not isinstance(x, (int, float)):
----> 6     raise TypeError('x must be numeric')
      7   elif x < 0:
      8     raise ValueError('x can not be negative')

TypeError: x must be numeric
```

Try …. Except block

- A suspicious code can be put in a try: block and use except: block to handle each kind of errors that can potentially arise in the code.

```
try:
   You do your operations here
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

```python
try:
  fp = open('sample.txt')
except IOError as e:
  print('Unable to open the file', e)

print("\n\n -------\n")

age = -1
while age <= 0:
  try:
    age = int(input('Enter your agen in years: '))
    if age <= 0:
      print('Your age must be positive')
  except(ValueError, EOFError):
    print('invalid response')

```

```
Unable to open the file [Errno 2] No such file or directory: 'sample.txt'


 -------

Enter your agen in years: -25
Your age must be positive
Enter your agen in years: Twenty Five
invalid response
Enter your agen in years: 25.2
invalid response
Enter your agen in years: 25
```

Raise An exception

```
def sqrt(x):
  if not isinstance(x, (int, float)):
    raise TypeError('x must be numeric')
  elif x < 0:
    raise ValueError('x cannot be negative')
  # do the real work here...
```

```
def sum(values):
  if not isinstance(values, collections.Iterable):
    raise TypeError('parameter must be an iterable type')
  total = 0
  for v in values:
    if not isinstance(v, (int, float)):
      raise TypeError('elements must be numeric')
    total = total+ v
  return total
```

Catch An exception

```
try:
  fp = open('sample.txt')
except IOError as e:
  print('Unable to open the file:', e)
```

```
age = −1                          # an initially invalid choice
while age <= 0:
  try:
    age = int(input('Enter your age in years: '))
    if age <= 0:
      print('Your age must be positive')
  except (ValueError, EOFError):
    print('Invalid response')
```

Assert() Function

- It is used to enforce a condition.
- It raises an exception when the condition is not met.

  `Assert Expression[, Arguments]`

- if Assertion fails, Python uses 'Arguments' for the **`AssertionError`** exception.

```
# Assertions

def KelvinToFahrenheit(Temperature):
    assert (Temperature >= 0),"Colder than absolute zero!"
    return ((Temperature-273)*1.8)+32

print (KelvinToFahrenheit(273))
print (int(KelvinToFahrenheit(505.78)))
print (KelvinToFahrenheit(-5))
```

```
32.0
451
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-30-e5becc09480c> in <module>()
      6 print (KelvinToFahrenheit(273))
      7 print (int(KelvinToFahrenheit(505.78)))
----> 8 print (KelvinToFahrenheit(-5))

<ipython-input-30-e5becc09480c> in KelvinToFahrenheit(Temperature)
      1
      2 def KelvinToFahrenheit(Temperature):
----> 3     assert (Temperature >= 0),"Colder than absolute zero!"
      4     return ((Temperature-273)*1.8)+32
      5

AssertionError: Colder than absolute zero!
```

Common Exception Types

Class	Description
Exception	A base class for most error types
AttributeError	Raised by syntax obj.foo, if obj has no member named foo
EOFError	Raised if "end of file" reached for console or file input
IOError	Raised upon failure of I/O operation (e.g., opening file)
IndexError	Raised if index to sequence is out of bounds
KeyError	Raised if nonexistent key requested for set or dictionary
KeyboardInterrupt	Raised if user types ctrl-C while program is executing
NameError	Raised if nonexistent identifier used
StopIteration	Raised by next(iterator) if no element; see Section 1.8
TypeError	Raised when wrong type of parameter is sent to a function
ValueError	Raised when parameter has invalid value (e.g., sqrt(−5))
ZeroDivisionError	Raised when any division operator used with 0 as divisor

Iterators & Generators

- An ***iterator*** is an object that manages an iteration through a series of values.

- If `i` is an iterator then, each call to the built-in function, `next(i)`, produces a subsequent element from the underlying series.

- A `StopIteration` exception raised to indicate that there are no further elements.

- An ***iterable*** is an object, obj, that produces an iterator via the syntax:

  ```
  > i = iter(obj).
  ```

- The most convenient technique for creating iterators in Python is through the use of ***generators***.

- Generators can have multiple yield commands

- Iterators and generators provide the benefit of ***lazy evaluation*** where the values are computed if requested, the entire series need not reside in memory at one time.

Example 1: Generator

```
# Generators

def factors1(n): # generator that computes factors
  for k in range(1,n+1):
    if n % k == 0: # divides evenly, thus k is a factor
      yield k

# an efficient version
def factors2(n): # generator that computes factors
  k = 1
  while k*k < n: # while k < sqrt(n)
    if n % k == 0:
      yield k
      yield n // k
    k += 1
  if k*k == n: # special case if n is perfect square
    yield k


num_factors = 0
f = []
#for factor in factors1(100):
for factor in factors2(100):
  if factor > 0:
    f.append(factor)
    num_factors += 1
print("Total number of factors:{}".format(num_factors))
print("Factors are:{}".format(f))
```

```
Total number of factors:9
Factors are:[1, 100, 2, 50, 4, 25, 5, 20, 10]
```

Example 2: Generator

```
# Fibpnacci series - You can produce infinite series

def fibonacci():
  a = 0
  b = 1
  while True: # keep going...
    yield a # report value, a, during this pass
    future = a + b
    a = b # this will be next value reported
    b = future

series =[]
for i in fibonacci():
  if(i < 100):
    series.append(i)
  else:
    break
print("Fibonacci series:{}".format(series))
```

```
Fibonacci series:[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Example 3: Iterator by using iter() command

```
# iterators
data = [1,2,3,4]

for i in iter(data):
  print(data[i-1])

for element in data:
  print(element)
```

```
1
2
3
4
1
2
3
4
```

Additional Python Conveniences

- Conditional Expressions:

expr1 **if** *condition* **else** *expr2*

```
if n >= 0:
    param = n
else:
    param = -n
result = foo(param)
```

```
result = foo(n if n >= 0 else -n)
```

- Comprehension Syntax: Produce one series of values based upon the processing of another series:

  - List Comprehension

```
result = [ ]
for value in iterable:
  if condition:
    result.append(expression)
```

```
[ expression for value in iterable if condition ]
```

```
squares = [ ]
for k in range(1, n+1):
  squares.append(k*k)
```

```
squares = [k*k for k in range(1, n+1)]
```

```
def factors(n):
  f = []
  for k in range(1,n+1):
    if n % k == 0:
      f.append(k)
  return f

print("Factors of 100: ", factors(100))
```

```
Factors of 100:  [1, 2, 4, 5, 10, 20, 25, 50, 100]
```

```
def factors(n):
  return [k for k in range(1,n+1) if n%k ==0]

print("Factors of 100: ", factors(100))
```

```
Factors of 100:  [1, 2, 4, 5, 10, 20, 25, 50, 100]
```

- Comprehension Syntax (continued)

`[ k*k for k in range(1, n+1) ]`	list comprehension
`{ k*k for k in range(1, n+1) }`	set comprehension
`( k*k for k in range(1, n+1) )`	generator comprehension
`{ k : k*k for k in range(1, n+1) }`	dictionary comprehension

- The generator syntax is particularly attractive when results do not need to be stored in memory.
- For example, to compute the sum of the first n squares, the generator syntax, total = sum(k k for k in range(1, n+1)), is preferred to the use of an explicitly instantiated list comprehension as the parameter

- Packing and Unpacking of Sequences:
    - ***Automated Packing:*** If a series of comma-separated expressions are given in a larger context, they will be treated as a single tuple, even if no enclosing parentheses are provided.

      $$\text{data} = 2,\ 4,\ 6,\ 8 \Rightarrow \text{tuple } (2,\ 4,\ 6,\ 8).$$

    - One common use of packing in Python is when **returning multiple values from a function**

      $$\textbf{return } \mathsf{x,\ y}$$

    - ***Automated Unpacking:*** Allowing one to assign a series of individual identifiers to the elements of sequence.

      $$\mathsf{a,\ b,\ c,\ d = range(7,\ 11)}$$

      This technique can be used to unpack tuples returned by a function.

      $$\mathsf{quotient,\ remainder = divmod(a,\ b)}$$

      This syntax can also be used in the context of a for loop, when iterating over a sequence of iterables:

      $$\textbf{for } \mathsf{x,\ y}\ \textbf{in}\ [\ (7,\ 2),\ (5,\ 8),\ (6,\ 4)\ ]:$$

      Iterate through dictionary items: $\textbf{for } \mathsf{k,\ v}\ \textbf{in}\ \mathsf{mapping.items()}:$

- ***Simultaneous Assignments***: Combination of automated packing & unpacking

```
x, y, z = 6, 2, 5
```

Swapping of values:

Example 1

```
temp = j
j = k
k = temp
```

⇨

```
j, k = k, j
```

Example 2

```
def fibonacci():
  a = 0
  b = 1
  while True:
    yield a
    future = a + b
    a = b
    b = future
```

⇨

```
def fibonacci():
  a, b = 0, 1
  while True:
    yield a
    a, b = b, a+b
```

Scopes & Namespaces

- The process of determining the value associated with an identifier is known as ***name resolution***.
- Whenever an identifier is assigned to a value, that definition is made with a
- specific ***scope.***
- Top-level assignments are typically made in what is known as ***global*** scope.
- Assignments made within the body of a function typically have scope that is local to that function call.
- Each distinct scope in Python is represented using an abstraction known as a ***namespace***.
- ***First-class objects*** are instances of a type that can be assigned to an identifier, passed as a parameter, or returned by a function. In Python, functions and classes are also treated as first-class objects.

Modules

- A ***module*** is a collection of closely related functions and
- classes that are defined together in a single file of source code which could be imported from within a program.
- Beyond the built-in definitions, the standard Python distribution includes perhaps tens of thousands of other values, functions, and classes that are organized in additional libra

```
from math import *,
import math
from math import pi, sqrt
```

- New modules can be created by putting relevant definitions in a file named with a .py suffix. Those definitions can be imported from any other .py file within the same project directory.

Existing Modules	
Module Name	**Description**
array	Provides compact array storage for primitive types.
collections	Defines additional data structures and abstract base classes involving collections of objects.
copy	Defines general functions for making copies of objects.
heapq	Provides heap-based priority queue functions (see Section 9.3.7).
math	Defines common mathematical constants and functions.
os	Provides support for interactions with the operating system.
random	Provides random number generation.
re	Provides support for processing regular expressions.
sys	Provides additional level of interaction with the Python interpreter.
time	Provides support for measuring time, or delaying a program.

Summary

We cover the following topics in this lecture:

- Introduction to Python Programming
- Various Sequence type Classes
- Operators
- Input/Output, file Handling
- Exception Handling
- Iterators & generators
- Python Conveniences
- Namespaces
- Modules