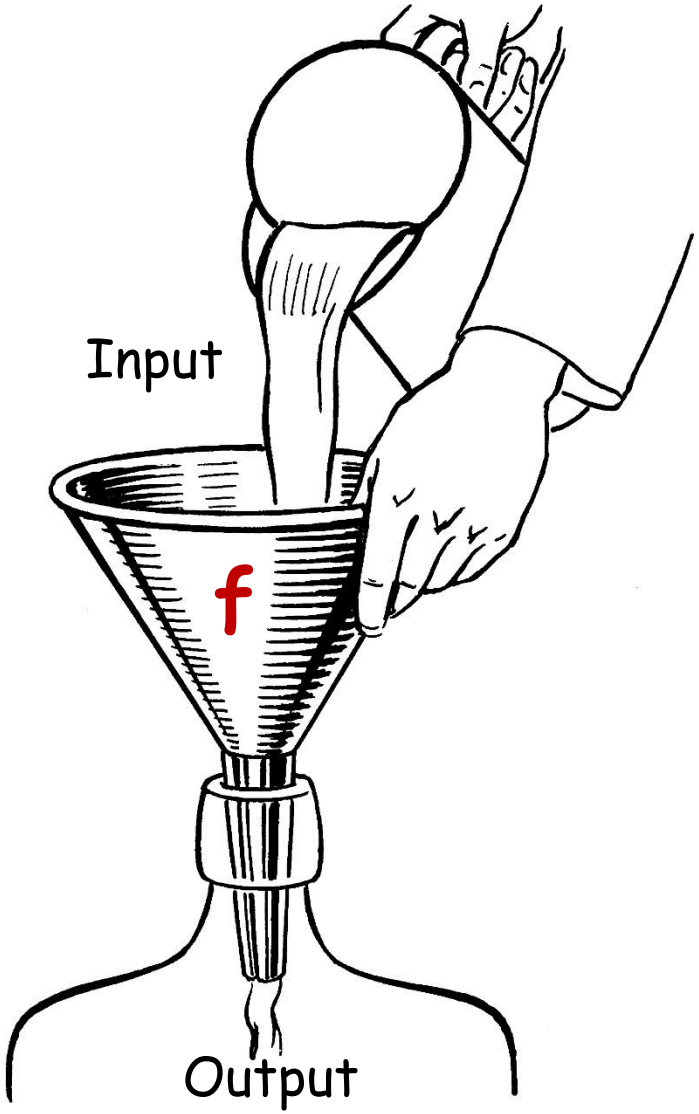
CS6.302 - Software System Development Monsoon 2025

Lab - 11: Python Session 2
Functions, Data Structures, File Handling, and Exception Handling

Programming using Python

f(unctions)

Parts of a function



Python built-in functions

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

To find out how they work:

https://docs.python.org/3.3/library/functions.html

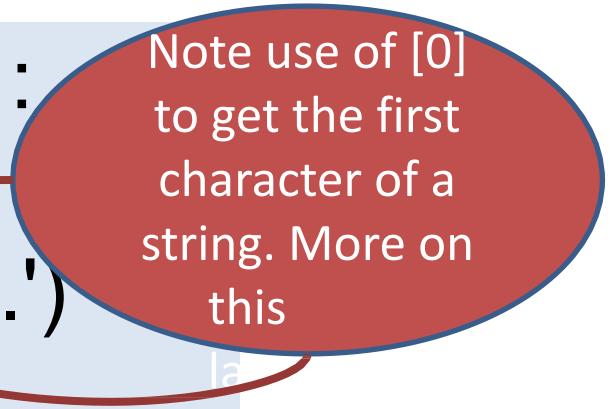
```
def max (a, b):
   "'return maximum among a and
   b"' if (a > b):
        return a
  else:
        return b
```

```
In[3] : help(max)
Help on function max in module  main :

max(a, b)
    return maximum among a and b
```

Keyword Arguments

```
def printName(first, last, initials) :
    if initials:
        print (first[0] + '. ' + last[0] + '.')
    else:
        print (first, last)
```



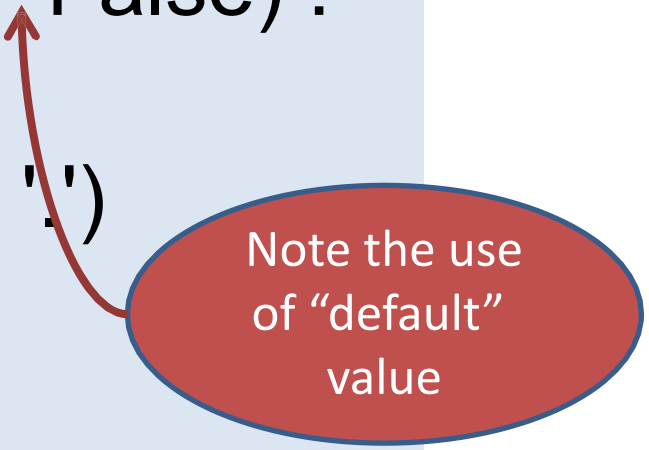
Call	Output
printName('Acads', 'Institute', False)	Acads Institute
printName('Acads', 'Institute', True)	A. I.
printName(last='Institute', initials=False, first='Acads')	Acads Institute
printName('Acads', initials=True, last='Institute')	A. I.

Keyword Arguments

- Parameter passing where formal is bound to actual using formal's name
- Can mix keyword and non-keyword arguments
  - All non-keyword arguments precede keyword arguments in the call
  - Non-keyword arguments are matched by position (order is important)
  - Order of keyword arguments is not important

Default

```
def printName(first, last, initials=False) :
    if initials:
        print (first[0] + '. ' + last[0] + '.')
    else:
        print (first, last)
```



Call	Output
printName('Acads', 'Institute')	Acads Institute
printName(first='Acads', last='Institute', initials=True)	A. I.
printName(last='Institute', first='Acads')	Acads Institute
printName('Acads', last='Institute')	Acads Institute

Default Values

- Allows user to call a function with fewer arguments
- Useful when some argument has a fixed value for most of the calls
- All arguments with default values must be at the end of argument list
  - non-default argument can not follow default argument

Globals

- Globals allow functions to communicate with each other indirectly
  - Without parameter passing/return value
- Convenient when two seemingly "far-apart" functions want to share data
  - No *direct* caller/callee relation
- If a function has to update a global, it must re-declare the global variable with global keyword.

Globals

```
PI = 3.14
def perimeter(r):
    return 2 * PI * r
def area(r):
    return PI * r * r

def update_pi():
    global PI
    PI = 3.14159
```

```
>>> print(area (100))
31400.0
>>> print(perimeter(10))
62.800000000000004
>>> update_pi()
>>> print(area(100))
31415.999999999996
>>> print(perimeter(10))
62.832
```

defines PI to be of float type with value 3.14.
PI can be used across functions. Any change to PI in update_pi will be visible to
all due to the use of global.

STRINGS
TUPLES
LISTS

Strings

- Strings in Python have type str
- They represent sequence of characters
  - Python does not have a type corresponding to character.
- Strings are enclosed in single quotes(') or double quotes(“)
  - Both are equivalent
- Backslash (\) is used to escape quotes and special characters

Strings

```
>>> name='intro to python'
>>> descr='acad\'s first course'
>>> name
'intro to python'
>>> descr
"acad's first course"
```

- More readable when print is used

```
>>> print descr
acad's first course
```

Length of a String

- len function gives the length of a string

```
>>> name='intro to python'
>>> empty=''
>>> single='a'
>>> len(name)
15
>>> len(single)
1
>>> len(empty)
0
>>> special='1\n2'
>>> len(special)
3
```

\n is a **single** character: the special character representing newline

Concatenate and Repeat

- In Python, + and * operations have special meaning when operating on strings
  - + is used for concatenation of (two) strings
  - * is used to repeat a string, an int number of time
  - Function/Operator Overloading

Concatenate and Repeat

```
>>> details = name + ', ' + descr
>>> details
"intro to python, acad's first course"

>>> print punishment
I won't fly paper airplanes in class

>>> print punishment*5
I won't fly paper airplanes in class
I won't fly paper airplanes in class
I won't fly paper airplanes in class
I won't fly paper airplanes in class
I won't fly paper airplanes in class
```

Quick question!

Do you see anything wrong with this block?

```
str1 = "which means it has even more than"
str2 = 76
str3 = "quirks"
print(str1 + str2 + str3)
```

```
---------------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call l
ast)
<ipython-input-2-3be15a6244a4> in <module>()
      2 str2 = 76
      3 str3 = " quirks"
----> 4 print(str1 + str2 + str3)

TypeError: must be str, not int
```

Another more generic way to fix it

```
str1 = "It has"
str2 = 76
str3 = "methods!"
print(str1, str2, str3)
```

```
It has 76 methods!
```

If we comma separate statements in a print function, we can have different variables printing!

Placeholders

- A way to interleave numbers is

```
pi = 3.14159 # Pi
d = 12756 # Diameter of eath at equator (in km)
c = pi*d # Circumference of equator

#Print using +, and casting
print("Earth's diameter at equator: " + str(d) + "km. Equator's circumference:" + str(c) + "km.")
#Print using several arguments
print("Earth's diameter at equator:", d, "km. Equator's circumference:", c, "km.")
#Print using .format
print("Earth's diameter at equator: {:.1f} km. Equator's circumference: {:.1f} km.".format(d, c))
```

```
Earth's diameter at equator: 12756km. Equator's circumference:40074.12204km.
Earth's diameter at equator: 12756 km. Equator's circumference: 40074.12204 km.
Earth's diameter at equator: 12756.0 km. Equator's circumference: 40074.1 km.
```

- Elegant and easy

Indexing

- Strings can be indexed
- First character has index 0

```
>>> name='Acads'
>>> name[0]
'A'
>>> name[3]
'd'
>>> 'Hello'[1]
'e'
```

Indexing

- Negative indices start counting from the right
- Negatives indices start from -1
- -1 means last, -2 second last, ...

```
>>> name='Acads'
>>> name[-1]
's'
>>> name[-5]
'A'
>>> name[-2]
'd'
```

Indexing

- Using an index that is too large or too small results in "index out of range" error

```
>>> name='Acads'
>>> name[50]

Traceback (most recent call last):
  File "<pyshell#136>", line 1, in <module>
    name[50]
IndexError: string index out of range
>>> name[-50]

Traceback (most recent call last):
  File "<pyshell#137>", line 1, in <module>
    name[-50]
IndexError: string index out of range
```

Slicing

- To obtain a substring
- s[start:end] means substring of s starting at index start and ending at index end-1
- s[0:len(s)] is same as s
- Both start and end are optional
  - If start is omitted, it defaults to 0
  - If end is omitted, it defaults to the length of string
- s[:] is same as s[0:len(s)], that is same as s

Slicing

```
>>> name='Acads'
>>> name[0:3]
'Aca'
>>> name[:3]
'Aca'
>>> name[3:]
'ds'
>>> name[:3] + name[3:]
'Acads'
>>> name[0:len(name)]
'Acads'
>>> name[:]
'Acads'
```

More Slicing

```
>>> name='Acads'
>>> name[-4:-1]
'cad'
>>> name[-4:]
'cads'
>>> name[-4:4]
'cad'
```

Understanding Indices for slicing

A	c	a	d	s	
0	1	2	3	4	5
-5	-4	-3	-2	-1	

Out of Range Slicing

A	c	a	d	s
0	1	2	3	4
-5	-4	-3	-2	-1

- Out of range indices are ignored for slicing
- when start and end have the same sign, if start >=end, empty slice is returned

Why?

```
>>> name='Acads'
>>> name[4:50]
's'
>>> name[40:50]
''
>>> name[-50:20]
'Acads'
>>> name[-50:-20]
''
>>> name[50:20]
''
>>> name [1:-1]
'cad'
```

Can we remove a character from string?

Tuples

- A tuple consists of a number of values separated by commas

```
>>> t = 'intro to python', 'amey karkare', 101
>>> t[0]
'intro to python'
>>> t[2]
101
>>> t
('intro to python', 'amey karkare', 101)
>>> type(t)
<type 'tuple'>
```

- Empty and Singleton Tuples

```
>>> empty = ()
>>> singleton = 1,   # Note the comma at the end
```

Nested Tuples

- Tuples can be nested

```
>>> course = 'Python', 'Amey', 101
>>> student = 'Prasanna', 34, course
>>> student
('Prasanna', 34, ('Python', 'Amey', 101))
```

- Note that course tuple is copied into student.
  - Changing course does not affect student

```
>>> course = 'Stats', 'Adam', 102
>>> student
('Prasanna', 34, ('Python', 'Amey', 101))
```

Length of a Tuple

- len function gives the length of a tuple

```
>>> course = 'Python', 'Amey', 101
>>> student = 'Prasanna', 34, course
>>> empty = ()
>>> singleton = 1,
>>> len(empty)
0
>>> len(singleton)
1
>>> len(course)
3
>>> len(student)
3
```

More Operations on Tuples

- Tuples can be concatenated, repeated, indexed and sliced

```
>>> course1
('Python', 'Amey', 101)
>>> course2
('Stats', 'Adams', 102)
>>> course1 + course2
('Python', 'Amey', 101, 'Stats', 'Adams', 102)
>>> (course1 + course2)[3]
'Stats'
>>> (course1 + course2)[2:7]
(101, 'Stats', 'Adams', 102)
>>> 2*course1
('Python', 'Amey', 101, 'Python', 'Amey', 101)
```

Unpacking Sequences

- Strings and Tuples are examples of sequences
  - Indexing, slicing, concatenation, repetition operations applicable on sequences
- Sequence Unpacking operation can be applied to sequences to get the components
  - *Multiple assignment* statement
  - LHS and RHS must have equal length

Unpacking Sequences

```
>>> student
('Prasanna', 34, ('Python', 'Amey', 101))
>>> name,roll,regdcourse=student
>>> name
'Prasanna'
>>> roll
34
>>> regdcourse
('Python', 'Amey', 101)
>>> x1,x2,x3,x4 = 'amey'
>>> print(x1,x2,x3,x4)
a m e y
```

Lists

- Ordered sequence of values
- Written as a sequence of comma-separated values between square brackets
- Values can be of different types

```
>>> lst = [1,2,3,4,5]
>>> lst
[1, 2, 3, 4, 5]
>>> type(lst)
<type 'list'>
```

same type

Lists

- List is also a sequence type
  - Sequence operations are applicable

```
>>> fib = [1,1,2,3,5,8,13,21,34,55]
>>> len(fib)
10
>>> fib[3] # Indexing
3
>>> fib[3:] # Slicing
[3, 5, 8, 13, 21, 34, 55]
```

Lists

- List is also a sequence type
  - Sequence operations are applicable

```
>>> [0] + fib # Concatenation
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> 3 * [1, 1, 2] # Repetition
[1, 1, 2, 1, 1, 2, 1, 1, 2]
>>> x,y,z = [1,1,2] #Unpacking
>>> print (x, y, z )
1 1 2
```

More Operations on Lists

- L.append(x)
- L.extend(seq)
- L.insert(i, x)
- L.remove(x)
- L.pop(i)
- L.pop()
- L.index(x)
- L.count(x)
- L.**sort**()
- L.**reverse**()

x is any value, seq is a sequence value (list, string, tuple, …),
i is an integer value

Mutable and Immutable Types

- Tuples and List types look very similar
- However, there is one major difference: Lists are **mutable**
  - Contents of a list can be modified
- Tuples and Strings are **immutable**
  - Contents can not be modified

Summary of Sequences

Operation	Meaning
seq[i]	i-th element of the sequence
len(seq)	Length of the sequence
seq1 + seq2	Concatenate the two sequences
num*seq seq*num	Repeat seq num times
seq[start:end]	slice starting from **start**, and ending at **end-1**
e in seq	True if e is present is seq, False otherwise
e not in seq	True if e is not present is seq, False otherwise
for e in seq	Iterate over all elements in seq (e is bound to one element per iteration)

Sequence types include String, Tuple and List.
Lists are mutable, Tuple and Strings immutable.

Programming with Python

Sets and Dictionaries

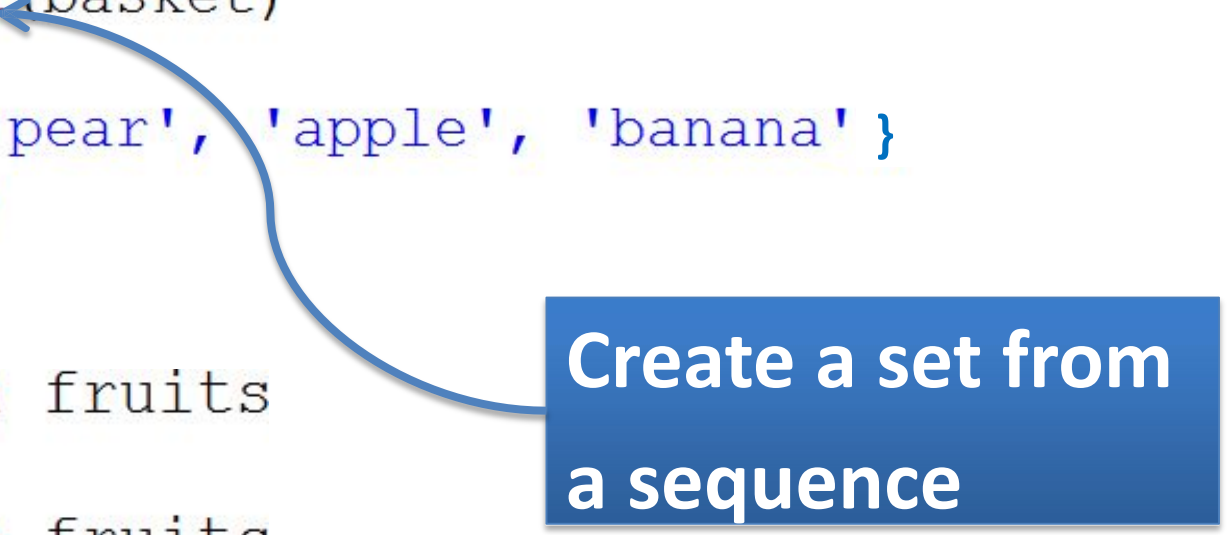
Sets

- An unordered collection with no duplicate elements
- Supports
  - membership testing
  - eliminating duplicate entries
  - Set operations: union, intersection, difference, and symmetric difference.

Sets

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'o
range', 'banana']
>>> fruits = set(basket)
>>> fruits
    {'orange', 'pear', 'apple', 'banana'}
>>> type(fruits)
        set

 >>> 'apple' in fruits
 True
 >>> 'mango' in fruits
 False
```



Set Operations

```
>>> A=set('acads')
>>> B=set('institute')
>>> A
    {'a', 's', 'c', 'd'])
>>> B
    {'e', 'i', 'n', 's', 'u', 't'}
>>> A - B # Set difference
    {'a', 'c', 'd'}
>>> A | B # Set Union
    {'a', 'c', 'e', 'd', 'i', 'n', 's', 'u', 't'}
>>> A & B # Set intersection
    {'s'])
>>> A ^ B # Symmetric Difference
set(['a', 'd', 'c', 'e', 't', 'i', 'u', 'n'])
```

Dictionaries

- Unordered set of *key:value* pairs,
- Keys have to be unique
- Key:value pairs enclosed inside curly braces {...}
- Empty dictionary is created by writing {}
- Dictionaries are mutable
  - add new key:value pairs,
  - change the pairing
  - delete a key (and associated value)

Operations on Dictionaries

Operation	Meaning
len(d)	Number of key:value pairs in d
d.keys()	List containing the keys in d
d.values()	List containing the values in d
k in d	True if key k is in d
d[k]	Value associated with key k in d
d.get(k, v)	If k is present in d, then d[k] else v
d[k] = v	Map the value v to key k in d (replace d[k] if present)
del d[k]	Remove key k (and associated value) from d
for k in d	Iterate over the keys in d

Operations on Dictionaries

```
>>> capital = {'India':'New Delhi', 'USA':'Washingto
n DC', 'France':'Paris', 'Sri Lanka':'Colombo'}
>>> capital['India'] # Get an existing value
'New Delhi'
>>> capital['UK'] # Exception thrown for missing key

Traceback (most recent call last):
  File "<pyshell#130>", line 1, in <module>
    capital['UK'] # Exception thrown for missing key
KeyError: 'UK'
>>> capital.get('UK', 'Unknown') # Use of default
value with get
'Unknown'
>>> capital['UK']='London' # Add a new key:val pair
>>> capital['UK'] # Now it works
'London'
```

Operations on Dictionaries

```
>>> capital.keys()
['Sri Lanka', 'India', 'UK', 'USA', 'France']
>>> capital.values()
['Colombo', 'New Delhi', 'London', 'Washington DC',
'Paris']
>>> len(capital)
5
>>> 'USA' in capital
True
>>> 'Russia' in capital
False
>>> del capital['USA']
>>> capital
{'Sri Lanka': 'Colombo', 'India': 'New Delhi', 'UK':
'London', 'France': 'Paris'}
```

Operations on Dictionaries

```
>>> capital['Sri Lanka'] = 'Sri Jayawardenepura Kotte' # Wikipedia told me this!
>>> capital
{'Sri Lanka': 'Sri Jayawardenepura Kotte', 'India': 'New Delhi', 'UK': 'London', 'France': 'Paris'}
```

```
>>> countries = []
>>> for k in capital:
        countries.append(k)
```

 # Remember: for ... in iterates over keys only

```
>>> countries.sort()
>>> countries                    # Sort values in a list
['France', 'India', 'Sri Lanka', 'UK']
```

Dictionary Construction

- The dict constructor: builds dictionaries directly from *sequences of key-value pairs*

```
>>> airports=dict([('Mumbai', 'BOM'), ('Delhi', 'Del
'),('Chennai', 'MAA'), ('Kolkata', 'CCU')])
>>> airports
{'Kolkata': 'CCU', 'Chennai': 'MAA', 'Delhi': 'Del',
'Mumbai': 'BOM'}
```

Programming with Python

File I/O

File I/O

- Files are persistent storage
- Allow data to be stored beyond program lifetime
- The basic operations on files are
  - open, close, read, write
- Python treat files as sequence of lines
  - sequence operations work for the data read from files

File I/O: open and close

open(filename, mode)

- While opening a file, you need to supply
  - The name of the file, including the path
  - The mode in which you want to open a file
  - Common modes are r (read), w (write), a (append)
- Mode is optional, defaults to r
- open(..) returns a file object
- close() on the file object closes the file
  - finishes any buffered operations

File I/O: Example

```
>>> players = open('tennis_players', 'w')
>>>
>>>
>>>
>>>
>>>
>>> players.close() # done with writing
```

- Do some writing
- How to do it?
  - see the next few slides

File I/O: read, write and append

- Reading from an open file returns the contents of the file
  - as **sequence** of lines in the program
- Writing to a file
  - IMPORTANT: If opened with mode 'w', **clears** the existing contents of the file
  - Use append mode ('a') to preserve the contents
  - Writing happens at the end

File I/O: Examples

```
>>> players = open('tennis_players', 'w')
>>> players.write('Roger Federar\n')
>>> players.write('Rafael Nadal\n')
>>> players.write('Andy Murray\n')
>>> players.write('Novak Djokovic\n')
>>> players.write('Leander Paes\n')
>>> players.close() # done with writing

>>> countries = open('tennis_countries', 'w')
>>> countries.write('Switzerland\n')
>>> countries.write('Spain\n')
>>> countries.write('Britain\n')
>>> countries.write('Serbia\n')
>>> countries.write('India\n')
>>> countries.write('India\n')
>>> countries.close() # done with writing
```

File I/O: Examples

```
>>> print(players)
<closed file 'tennis_players', mode 'w' at 0x
031A48B8>
>>> print(countries)
<closed file 'tennis_countries', mode 'w' at
0x031A49C0>

>>> n = open('tennis_players', 'r')
>>> c = open('tennis_countries', 'r')
>>> n
<open file 'tennis_players', mode 'r' at 0x03
1A4910>
>>> c
<open file 'tennis_countries', mode 'r' at 0x
031A4A70>
```

```
>>> pn = n.read() # read all players
>>> pn
'Roger Federar\nRafael Nadal\nAndy Murray\nNo
vak Djokovic\nLeander Paes\n'
>>> print(pn)
Roger Federar
Rafael Nadal
Andy Murray
Novak Djokovic
Leander Paes

>>>
>>> n.close()
```

Note empty line due to '\n'

File I/O: Examples

```
>>> n = open('tennis_players', 'r')
>>> c = open('tennis_countries', 'r')
>>> pn, pc = [], []
>>> for l in n:
    pn.append(l[:-1]) # ignore '\n'

>>> n.close()
>>> for l in c:
    pc.append(l[:-1])
>>> c.close()
```

Note the use of for ... in for sequence

```
>>> print(pn, '\n', pc)
['Roger Federar', 'Rafael Nadal', 'Andy Murra
y', 'Novak Djokovic', 'Leander Paes']
['Switzerland', 'Spain', 'Britain', 'Serbia',
'India', 'India']
```

File I/O: Examples

```
>>> name_country = []
>>> for i in range(len(pn)):
        name_country.append((pn[i], pc[i]))


>>> print(name_country )
[('Roger Federar', 'Switzerland'), ('Rafael N
adal', 'Spain'), ('Andy Murray', 'Britain'),
('Novak Djokovic', 'Serbia'), ('Leander Paes'
, 'India')]
>>> n2c = dict(name_country)
>>> print(n2c)
{'Roger Federar': 'Switzerland', 'Andy Murray
': 'Britain', 'Leander Paes': 'India', 'Novak
Djokovic': 'Serbia', 'Rafael Nadal': 'Spain'}
>>> print(n2c['Leander Paes'])
India
```

Error Handling

- Avoid crashing of program
- Handle errors graciously

```
import csv
path = 'C:/Users/Abhinav/Desktop/IIIT_Pyton/missing.csv'
f = open(path, 'r')
row = csv.reader(f)
header = next(row)
for line in row:
    try:
        part1 = line[2].strip("'")
        part2 = line[3].strip("'")
    except ValueError as err:
        print("Bad Data: ", err)
        continue
    partM = int(part1) * float(part2)
    print(partM)
f.close()
```

Argument syntax

Variable and keyword arguments

```
def function(name, address="abcd"):
```

Arbitrary arguments (non-keyword)

```
def hypervolume(*length):
    a = 1
    for v in length:
        a *= v
    print(a)
```

Keyword arguments

```
def tag(name, **kwargs):
    print(name)
    print(kwargs)
tag('img', src="iiir.jpg", alt="knowledge",border=1)
```

Lambda Function

- **lambda** function is called an anonymous function. It is a single expression with implicit return.
- It can have any number of arguments but only one expression

lambda arguments : expression

```
d = {'apple': 18, 'orange': 20, 'banana': 5, 'rotten tomato': 1}
sorted(d.items(), key=lambda x: x[1])
```

High Order Functions

- We can use functions as data (objects) same as int, string, float etc.
- This is very useful when people write code that take functions as input

```
>>> def add(x,y):
        def add_closure():
                print('Adding {} + {} = {}'.format(x,y,x+y))
                return x+y
        return add_closure

>>> a = add(2,3)
>>> a()
Adding 2 + 3 = 5
5
```

```
>>> import time
>>> def after(second,func):
        time.sleep(second)
        func()


>>> def hello():
        print("Hello World")


>>> after(5,hello)
Hello World
```

- Closures : add_closure() do not take x,y. It just captures these x, y. This is called closure.

Programming with Python

Exception Handling

Introduction to Exception Handling

What is an Exception?

An **exception** is an error that occurs during the execution of a program. Whan an error happens, Python generates an exception that, if not handled, will cause the program to crash.

- **Why handle them?** To prevent the program from stopping unexpectedly and to manage errors gracefully, perhaps by giving the user a helpful message.

Introduction to Exception Handling

The ‘try’ and ‘except’ Block

The fundamental way to handle exceptions is with a ‘try…except’ block.

- The code that might cause an error is placed in the ‘try’ block.
- The code to execute if an error occurs is placed in the ‘except’ block.

Introduction to Exception Handling

Example: ‘ZeroDivisionError’
Let’s see what happens when we try to divide by zero.

```
numerator = 10
denominator = 0

try:
    # This line might cause an error
    result = numerator / denominator
    print(result)
except ZeroDivisionError:
    # This block runs only if a ZeroDivisionError occurs
    print("Oops! You can't divide a number by zero.")

print("The program continued without crashing!")
```

Common Exceptions: ‘TypeError’, ‘ValueError’, ‘FileNotFoundError’, IndexError’.

Advanced Exception Handling: ‘else’ and ‘finally’

The ‘else’ Clause

You can add an optional ‘else’ clause after the ‘except’ block. The code inside the ‘else’ block will run **only if no exceptions were raised** in the ‘try’ block.

The ‘finally’ Clause

The ‘finally’ clause is also optional. The code inside the ‘finally’ block will **always by executed**, no matter if an exception occurred or not. This is extremely useful for “cleanup” actions, like closing a file.

Complete Example: File Handling

This example combines ‘try’, ‘except’, ‘else’, and ‘finally’ for a common task: reading a file.

```
try:
    # Attempt to open and read a file
    f = open('my_data.txt', 'r')
    content = f.read()

except FileNotFoundError:
    # This runs if 'my_data.txt' doesn't exist
    print("Error: The file could not be found.")

except Exception as e:
    # A general catch-all for any other exceptions
    print(f"An unexpected error occurred: {e}")

else:
    # This runs ONLY if the try block was successful
    print("File read successfully!")
    print("--- File Content ---")
    print(content)

finally:
    # This ALWAYS runs, ensuring the file is closed
    if 'f' in locals() and not f.closed:
        f.close()
    print("--- Cleanup finished. ---")
```