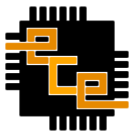
ECE 364
Software Engineering Tools Laboratory

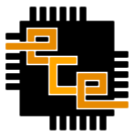
Lecture 4

Python: Collections I



Lecture Summary

- Lists
- Tuples
- Sets
- Dictionaries
- Printing, More I/O
- Bitwise Operations

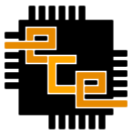


Lists

- `list` is a built-in Python data type
  - Much more powerful than plain old arrays
  - Can also be used to implement stacks and queues

- Lists are containers of things (objects)
  - Items need not be of the same data type

```
A = [1, 2, 3, 4]
B = [1, "Big Deal", [1, 2], 6.7J]
```



Lists (2)

- Lists are mutable, elements can be reassigned:

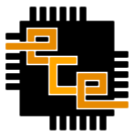
```
A = [1, 2, 3]
A[0] = “First”
```

- Use the `len(X)` function to return the length of a list

```
len(A)              # Returns 3
```

- Lists are **not** sparse – an index must exist

```
A[9] = "foo" # Illegal - causes a runtime error
```

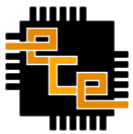


Indexing

Negative indices are allowed in Python

```
X = ["1st", "2nd", "3rd"]
X[0] = X[-3] = "1st"
X[1] = X[-2] = "2nd"
X[2] = X[-1] = "3rd”
```

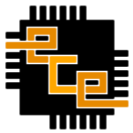
- 0 is the index of the leftmost item
- -1 is the index of the rightmost item



Slicing

- Slicing is a way to extract multiple elements from lists, tuples and strings.

`A[M:N]`	A slice of elements starting from index `M` and ending at index `N-1`
`A[M:N:S]`	A slice of elements starting from index `M` and ending at index `N-1`, with a step `S`
`A[M:]`	A slice of elements starting from index `M`
`A[:N]`	A slice of elements starting from index `0` and ending at index `N-1`
`A[:]`	A slice containing all elements of A

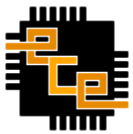


Slicing (2)

- Many things in Python can be sliced.
  - List, tuples and strings just to name a few

```
A = [1, 2, 3, 4, 5]
B = "ECE 364 is only 1 credit hour."
```

```
A[2:4] is [3, 4]        B[4:7] is '364'

A[:3] is [1, 2, 3]
```

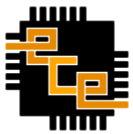


Tuples

- `tuple` is essentially an immutable list
  - Once created the contents can not be changed.
  - You can read using indexing and slicing.

- Basic Syntax

```
A = (1, "Big Deal", [1, 2], 6.7J)
A[0] is 1
A[1] is "Big Deal"
```



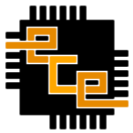
Tuples (2)

- To create a tuple from a list use `tuple()`

```
A=[1,2,3]
B=tuple(A)    # B is (1, 2, 3)
```

- To create a tuple from a string use `tuple()`

```
S="Hello"
T=tuple(S)
# T is ('H','e','l','l','o')
```



Tuples (3)

- Tuples can be unpacked.

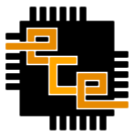
```
x = 2; y = 3
point = (x, y)
z, w = point                # z = 2 and w = 3
```

- This is extremely useful in iterations.

```
namesAndAges = [('Alex', 'Gheith', 40),
                ('Mary', 'Hanson', 22),
                ('John', 'Stewart', 33)]

for first, last, age in namesAndAges:
    # Do something.
```

- You can also choose not to use all elements in the tuple. Note that () are optional:

```
for first, _, age in namesAndAges:
    # Do something.
```



Sets

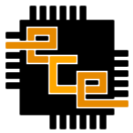
- A `set` is an unordered collection with no duplicate elements.

```
grades = {'A', 'D', 'B', 'C', 'D', 'B', 'A', 'D',
          'C', 'D', 'B', 'A', 'D'}
# grades = {'C', 'D', 'A', 'B'}
```

- Used for fast membership testing and maintaining unique elements.

```
grades = {'C', 'D', 'A', 'B'}
'C' in grades # Answer is True
'F' in grades # Answer is False
```

- Support mathematical operations like union (`|`), intersection (`&`), difference (`-`), and symmetric difference (`^`).

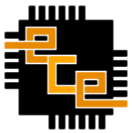


More on Strings

- Strings can be viewed as lists, and hence support list functions.

- However, strings are **immutable** and can not be changed

- String functions that perform formatting, whitespace removal etc. are creating new copies of the original string



More on Strings (2)

`substr in StrVar`

returns `True` if `substr` is in the list, `False` if it is not

`substr not in StrVar`

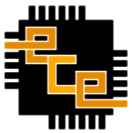
returns `True` if `substr` is not in the list, `False` if it is

`StrVar.find(substr)`

returns the index of the first occurrence of `substr` or `-1` if not found

`StrVar.rfind(substr)`

like `find()` but begins searching at the end of the string



More on Strings (3)

`StrVar.count(substr)`

returns the number of times `substr` occurs in the string, or `0` if not found

`StrVar.endswith(substr)`

returns `True` if `StrVar` ends with `substr`

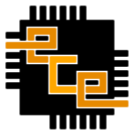
`StrVar.startswith(substr)`

returns `True` if `StrVar` ends with `substr`

`stringVar.replace(p,q,n)`

Replaces `n` occurrences of the substring `p` with the string `q`.

`n` is optional, default behavior is to replace all matches.



More on Strings (4)

`StrVar.isalnum()`
    returns `True` if the string has only alphanumeric characters

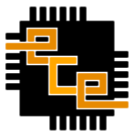
`StrVar.isalpha()`
    returns `True` if the string has only alpha characters

`StrVar.isdigit()`
    returns `True` if the string has only digits

`StrVar.isspace()`
    returns `True` if the string has only whitespace

`StrVar.isupper()`
    returns `True` if the string has only uppercase characters

`StrVar.islower()`
    returns `True` if the string has only lowercase characters



More on Strings (5)

`StrVar.lower()`

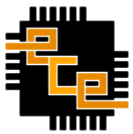
returns a copy of the string converted to lower case

`StrVar.upper()`

returns a copy of the string converted to upper case

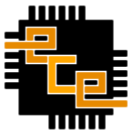
`StrVar.title()`

returns a copy of the string converted to title case

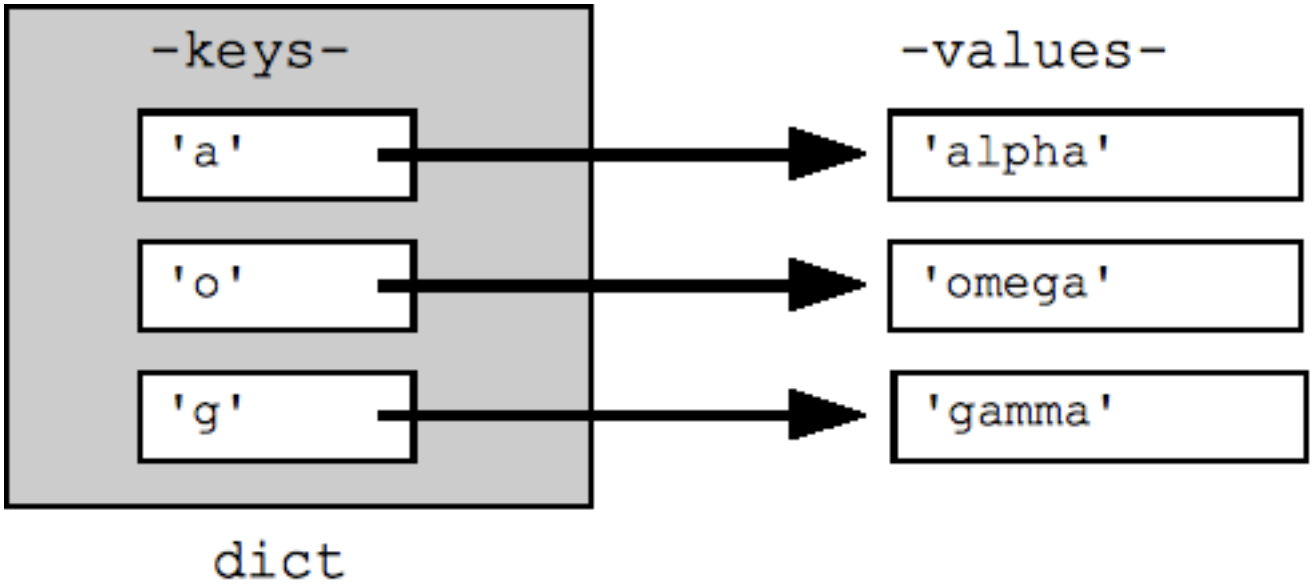


Dictionaries

- A dictionary is an unordered associative container that maps keys to values
- Dictionary is also called "Map" and "Lookup-Table" in other languages.
- A key can be any immutable value
  - Integers, strings, tuples etc.
- A value can be any type
  - Integers, strings, tuples, lists, dictionary etc.
- Items in a dictionary always exist as key-value pairs



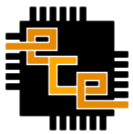
Dictionaries (2)



Source: http://code.google.com/edu/languages/google-python-class/dict-files.html

Dictionaries (3)

```
# To crate an empty dictionary
A = {}

# To create an empty set, you have use:
# s = set()

# To set initial key:value pairs
B = {'a' : 'alpha', 'o' : 'omega', 'g' : 'gamma'}

# Note that keys do NOT have to be of the same type
C = {(1,2) : True, "foo" : [1, 2, 3], 3.14 : "pi"}
```



Dictionaries (4)

- Dictionary values are accessed by specifying a key

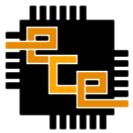
```
A[Key] # Gets the value associated with Key
```

For example:

```
myMap = {'a' : 'alpha', 'o' : 'omega', 'g' : 'gamma'}
l = myMap['a'] # The value in l is 'alpha'
```

- If a `key:value` pair is not present a `KeyError` exception is raised

```
g = myMap['b'] # This will raise a KeyError
```



Dictionaries (5)

- To check if an item exists in a dictionary:

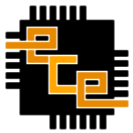
```
Key in A                # True if Key in A
```

- To negate the test:

```
Key not in A            # True if Key is not in A
```

- In Python 2.x, there was a function called `has_key` that has been removed in Python 3.x

```
A.has_key(key)                # True if Key is in A
```

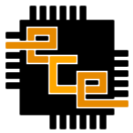


Dictionaries (6)

- `get(<Key>, <NotFoundValue>)`
  - returns `<NotFoundValue>` instead of raising an exception if `<Key>` is not found in the dictionary
  - `<NotFoundValue>` has a default value of `None`

```
A = {"red":23, "green":42}
```

`A.get("red")` returns `23`
`A.get("blue")` returns `None`
`A.get("red", "Not Found")` returns `23`
`A.get("blue", "Not Found")` returns `"Not Found"`
`A.get("blue", (1, 2, 3))` returns `(1, 2, 3)`
`A.get((1, 2), 0)` returns `0`



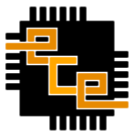
Dictionaries (7)

- To insert or change an item:

```
A[Key] = Value
```

- To merge two dictionaries use `update()`

```
>>> A = {1:20, -5:7, 8.2:31}
>>> B = {1:'foobar', 9:0}
>>> A.update(B)
>>> print(A)
{1: 'foobar', -5: 7, 9: 0, 8.2: 31}
```



Dictionaries (8)

- To delete an item from a dictionary:

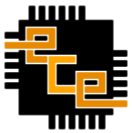
```
del A[Key] # does not return a value!
```

- To remove an item and get the value:

```
Value = A.pop(Key)
```

- To remove and item and get both the key and value:

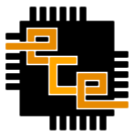
```
key, value = A.popitem() # does not take a Key!
```



Dictionaries (9)

- To obtain a list of the keys in a dictionary use the `keys()` function:

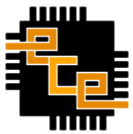
```
>>> A = {   "Foobar" : 100,
...                   2 : "Big Deal",
...         (1, 2, 34) : [[1, 2], "Yuk"]}

>>> print(A.keys())
[(1, 2, 34), 2, 'Foobar']
```



Dictionaries (10)

- To obtain a list of the values in a dictionary use the `values()` function:

```
>>> A = {   "Foobar" : 100,
...                   2 : "Big Deal",
...         (1, 2, 34) : [[1, 2], "Yuk"]}

>>> print(A.values())
[[1, 2], 'Yuk'], 100, 'Big Deal']
```

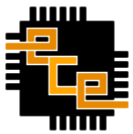


Dictionaries (11)

- To get a list of `key:value` pairs use the `items()` function

- Returns a list of `(key, value)` tuples

```
>>> A = {"a" : "alpha",
...      "b" : "big",
...    (1,2) : True }

>>> print(A.items())
  [("a", "alpha"), ("b", "big"), ((1,2), True)]
```



Dictionaries (12)

- In a for loop, a dictionary returns its keys:

```
for key in A:
    ...
```

- An equivalent statement would be:

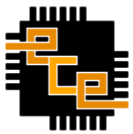
```
for key in A.keys():
    ...
```

- To iterate over its values only, use:

```
for value in A.values():
    ...
```

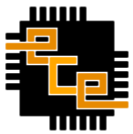
- To iterate over values and keys, use:

```
for key, value in A.items():
    ...
```



File Attribute Testing

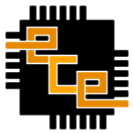
- Python provides functions to test file attributes in the `os` module
- `os.access(Path, Attribute)`
  - `Path` – String file path
  - `Attributes` – Flags
    - `os.R_OK` – File is readable
    - `os.W_OK` – File is writeable
    - `os.X_OK` – File is executable



File Attribute Testing (2)

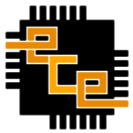
- File attributes are actually just numbers so you can combine them with bitwise operators

```
if os.access(file, os.R_OK):
  print("{} is readable!".format(file))

if os.access(file, os.R_OK | os.X_OK):
  print("{} is both readable and executable!"
  .format(file))
```



File Attribute Testing (3)

- Other helpful functions from the `os` module check properties of file paths
  - `os.path.exists(path)`
  - `os.path.isfile(path)`
  - `os.path.isdir(path)`



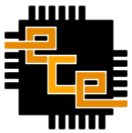
File I/O

- The `open()` function opens a file and returns a special object that represents the file
  - Very much like a `FILE*` pointer from C
  - Raises an exception when the file is not found

```
FileObject = open(FileName, Mode)
# Do some work.
FileObject.close()
```

- Modes:
  `"r"` - open for reading
  `"w"` - erase file and open for writing
  `"a"` - open file and append to end for writing

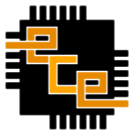
- This is NOT the preferred method in this lab.



File I/O (2)

- The preferred method to open files is using the `with` keyword.
- The `with` keyword is a shorthand for a lot of work in the background to ensure resources are claimed by the system when done, i.e. no need to invoke `fileObj.close()`.
- Can be used for both reading and writing.
- Note that once you read the file content, you should leave the “with” block.

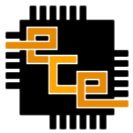
```
# The "myFile" below is called the file alias.
# This is called a "with-block"

with open('textFile.txt', 'r') as myFile:
    all_lines = myFile.readlines()

# The variable 'all_lines' is now populated.

for line in all_lines:
    # Do something
```



Command Line Arguments

- The `sys` module provides access to program arguments

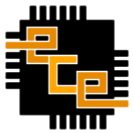
- `sys.argv` is the list of command line arguments passed to your script
  - `sys.argv[0]` is the same as `$0` from Bash
  - Arguments are passed as strings so you may need to convert!



Command Line Arguments (2)

```
import sys

total = 0

# Loop over arguments 1 to N
# Why not include the 0th arg?

for arg in sys.argv[1:]:
    total += float(arg)

print("The sum is {:f}".format(total))
```

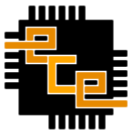
Hint: Sum of list element can be obtained using the `sum()` *function.*



Reading from `stdin`

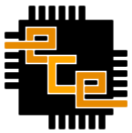
- `sys.stdin.readline()`
  - Read a single line from `stdin`
  - Will include the `\n` at the end of the line!
  - Returns the empty string at the end of input

- `input([prompt])`
  - Read a single line from `stdin`
  - Will strip the `\n` at the end of the line.
  - `[prompt]` is an optional prompt string



Reading from stdin (2)

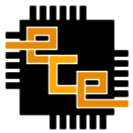
```
import sys

s=sys.stdin.readline()

# empty string will evaluate to False
while s:

    # remove the extra \n at the end
    print(s.rstrip())

    # read next line
    s=sys.stdin.readline()
```



Reading from `stdin` (3)

- `sys.stdin.readlines()`
  - Reads every line from `stdin` and returns a list containing each line
  - `\n` is still included on each line!

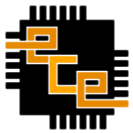
```
lines=sys.stdin.readlines()
for L in lines:
      L = L.rstrip()
      print(L)
```



Reading from `stdin` (4)

- A for loop can be used to read the entire contents of a file stream

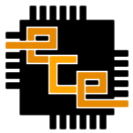
```
# You can "loop over" file streams!
for line in sys.stdin:
  line = line.rstrip()
  print(line)
```



Data Pretty Printer

- A quick way to printout the content of a collection is using the Data Pretty Printer module.
- Try out the following code:

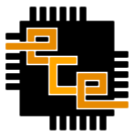
```
from pprint import pprint as pp

# Create a large dictionary:
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
        'Saturday', 'Sunday']
occurrences = [32, 12, 67, 21, 9, 45, 83]
dict_example = {day: occurrence for day, occurrence in
                zip(days, occurrences)}

# Regular Printing
print(dict_example)

# Pretty Printing
pp(dict_example)
```



Expressing Numbers in Base 2/8/10/16

- Use the following formats to express numbers in different bases

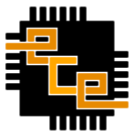
Base	Name	Format	Examples
2	Binary	0b<digits>	0b1010 0b11111111
8	Octal	0o<digits>	0o112 -0o5534563
10	Decimal	<digits>	123 -17890423
16	Hexadecimal	0x<digits>	0xdeadbeef 0x1234abcd



Numbers to String

- If you want to get a string representation of a number in a specific base

Base	Name	Function	Examples
2	Binary	bin(x)	bin(10) -> '0b1010' bin(0x1c) -> '0b11100'
8	Octal	oct(x)	oct(10) -> '0o12' oct(0b11100) -> '0o34'
10	Decimal	str(x)	str(10) -> '10' str(034) -> '28'
16	Hexadecimal	hex(x)	hex(128) -> '0x80' hex(0b10111) -> '0x17'



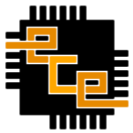
Bitwise Operators

- Left Shift – Shift each bit to the left by one position, shifts in zero to the leftmost position

```
A << n    # left shift A by n places
```

- Right Shift – Shift each bit to the right by one position, shifts in 1 to the leftmost position if the number is negative, 0 otherwise

```
A >> n    # right shift A by n places
```



Bitwise Operators (2)

- and – Perform a bit by bit “and” of two numbers. If each bit is set to 1 then set the output bit is set to 1, otherwise 0

```
A & B
```

- or – Perform a bit by bit “or” of two numbers. If either bit is set to 1 then set the output bit to 1, otherwise 0

```
A | B
```

- xor – Perform a bit by bit “exclusive or” of two numbers. Sets the output bit to 1 if one of the corresponding bits is set to 1 but not both

```
A ^ B
```

- complement – Flip the value of each bit from 1 to 0 or 0 to 1

```
~A
```