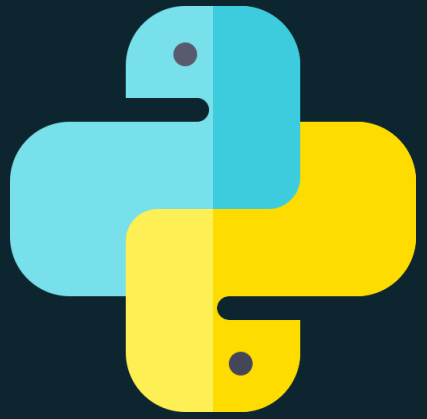
Brief Python

Python Course for Programmers



Learn Python Language for Data Science

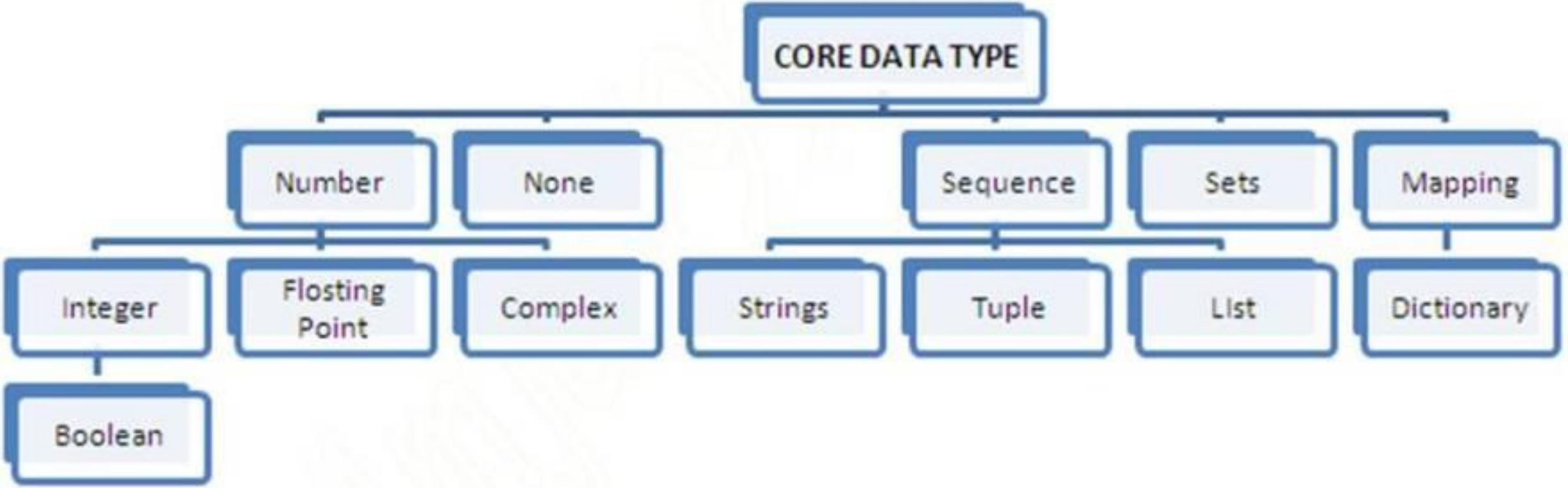
CHAPTER 3: DATA COLLECTION

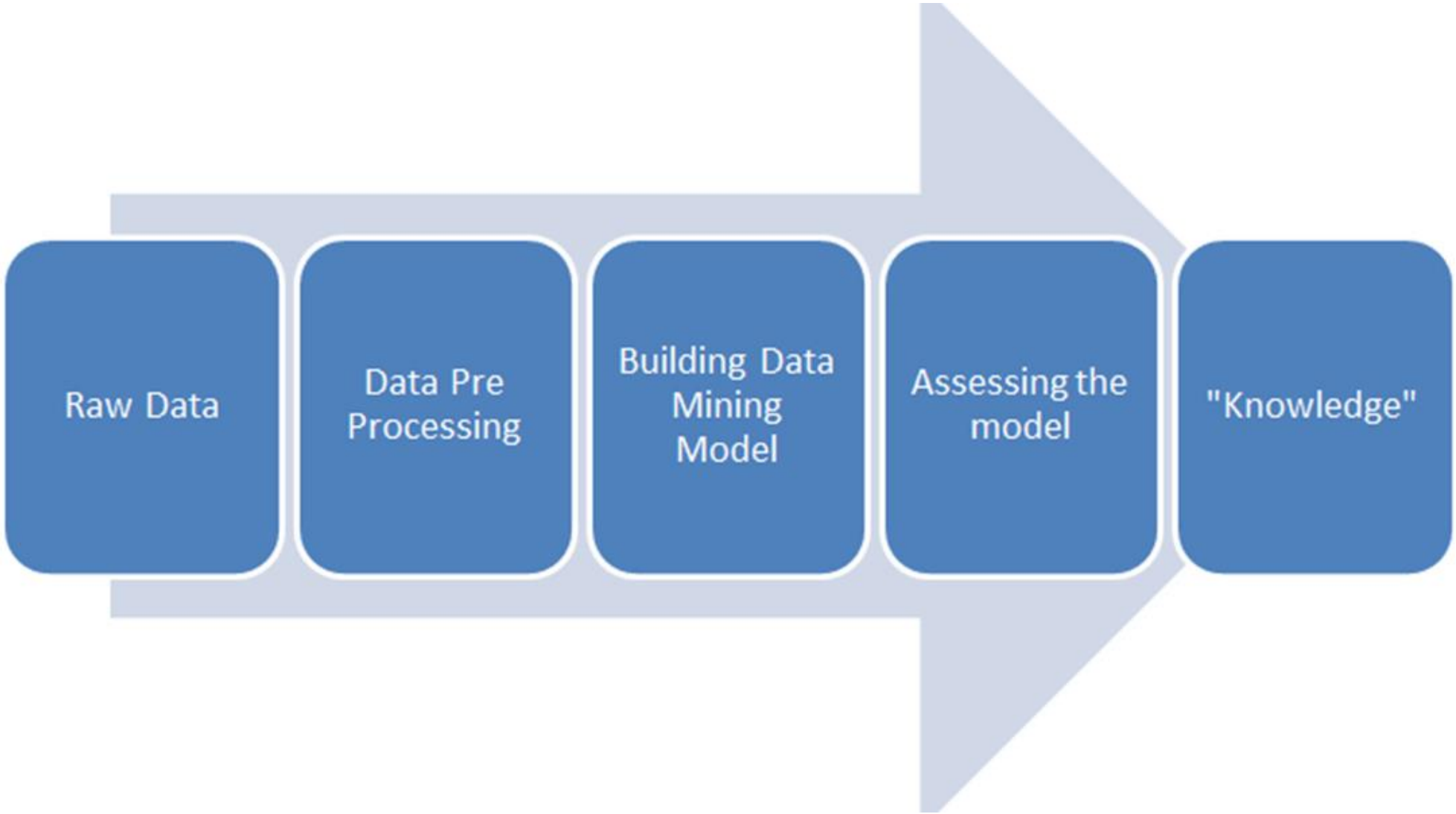
DR. ERIC CHOU

IEEE SENIOR MEMBER

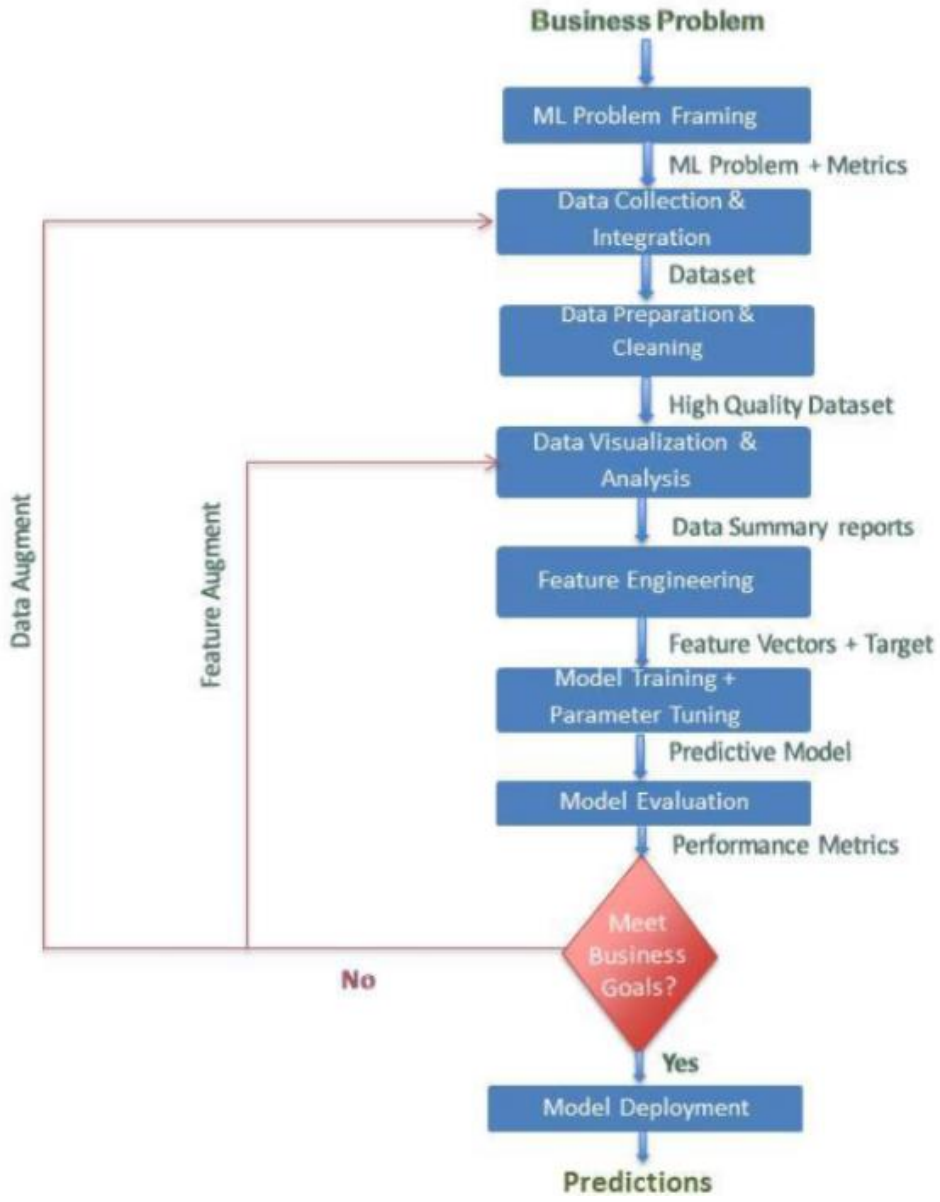
Objectives

- Study the built-in Python Basic Data Structure
  - String
  - Tuple
  - List
  - Set
  - Dictionary


CORE DATA TYPE
Number
None
Sequence
Sets
Mapping
Integer
Flosting Point
Complex
Strings
Tuple
List
Dictionary
Boolean


Raw Data
Data Pre Processing
Building Data Mining Model
Assessing the model
"Knowledge"

Model Building Process




WEB SERVER
NGINX
BACKEND LANGUAGE
php
Java
node
python
Scala
DATABASE
mongoDB
redis
MySQL
PostgreSQL
jQuery
zepto.js
HTML CSS JS
INTERNET
CLIENT SIDE
SERVER SIDE

String

LECTURE 1

String Data Type

- A string is a sequence of characters
- A string literal uses quotes 'Hello' or "Hello"
- For strings, + means “concatenate”
- When a string contains numbers, it is still a string
- We can convert numbers in a string into a number using int()

```
>>> str1 = "Hello"
>>> str2 = 'there'
>>> bob = str1 + str2
>>> print(bob)
Hellothere
>>> str3 = '123'
>>> str3 = str3 + 1
Traceback (most recent call
last):  File "<stdin>", line 1,
in <module>
TypeError: cannot concatenate
'str' and 'int' objects
>>> x = int(str3) + 1
>>> print(x)
124
>>>
```

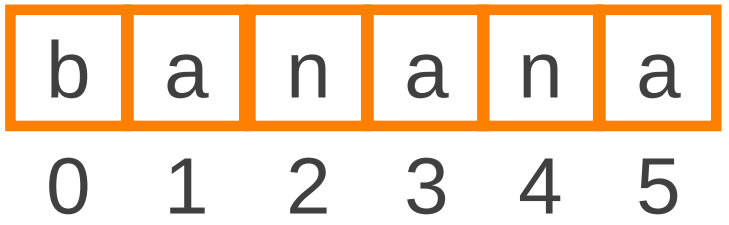
Reading and Converting

- We prefer to read data in using strings and then parse and convert the data as we need
- This gives us more control over error situations and/or bad user input
- Input numbers must be converted from strings

```
>>> name = input('Enter:')
Enter:Chuck
>>> print(name)
Chuck
>>> apple = input('Enter:')
Enter:100
>>> x = apple - 10
Traceback (most recent call
last):  File "<stdin>", line 1,
in <module>
TypeError: unsupported operand
type(s) for -: 'str' and 'int'
>>> x = int(apple) - 10
>>> print(x)
90
```

Looking Inside Strings

- We can get at any single character in a string using an index specified in square brackets
- The index value must be an integer and starts at zero
- The index value can be an expression that is computed




```
>>> fruit = 'banana'
>>> letter = fruit[1]
>>> print(letter)
a
>>> x = 3
>>> w = fruit[x - 1]
>>> print(w)
n
```

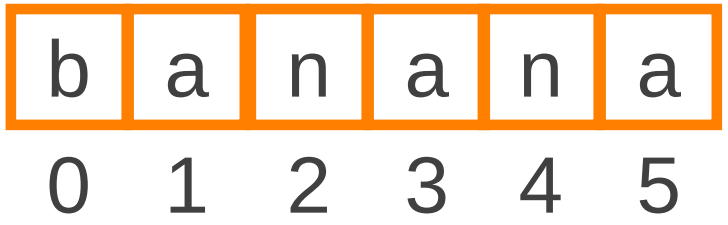
A Character Too Far

- You will get a python error if you attempt to index beyond the end of a string
- So be careful when constructing index values and slices

```
>>> zot = 'abc'
>>> print(zot[5])
Traceback (most recent call
last):  File "<stdin>", line
1, in <module>
IndexError: string index out
of range
>>>
```

Strings Have Length

- The built-in function len gives us the length of a string

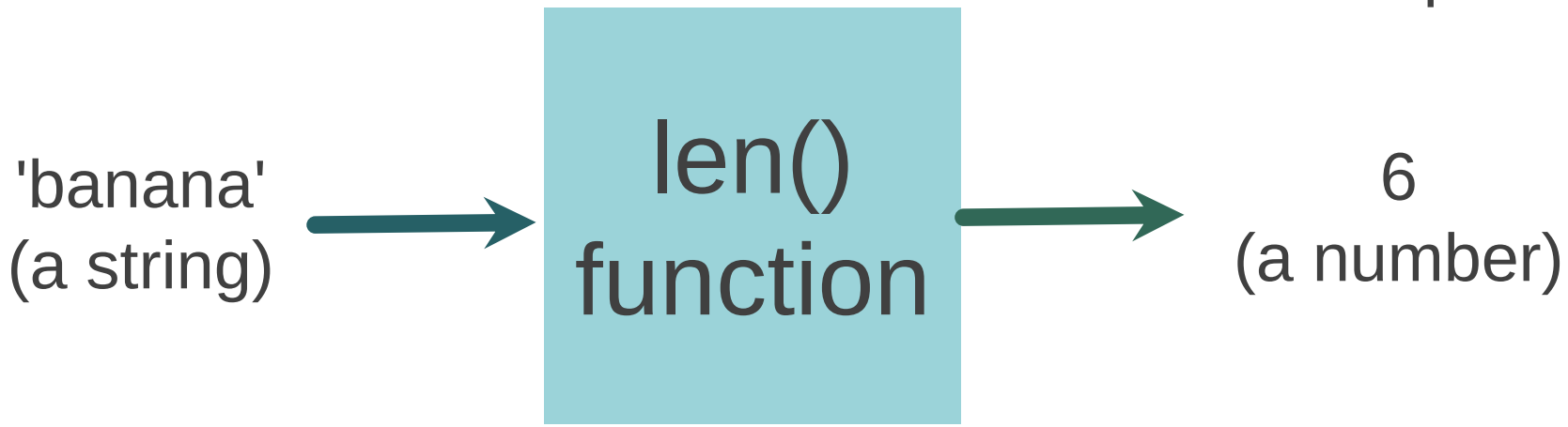



```
>>> fruit = 'banana'
>>> print(len(fruit))
6
```

len Function

```
>>> fruit = 'banana'
>>> x = len(fruit)
>>> print(x)
6
```

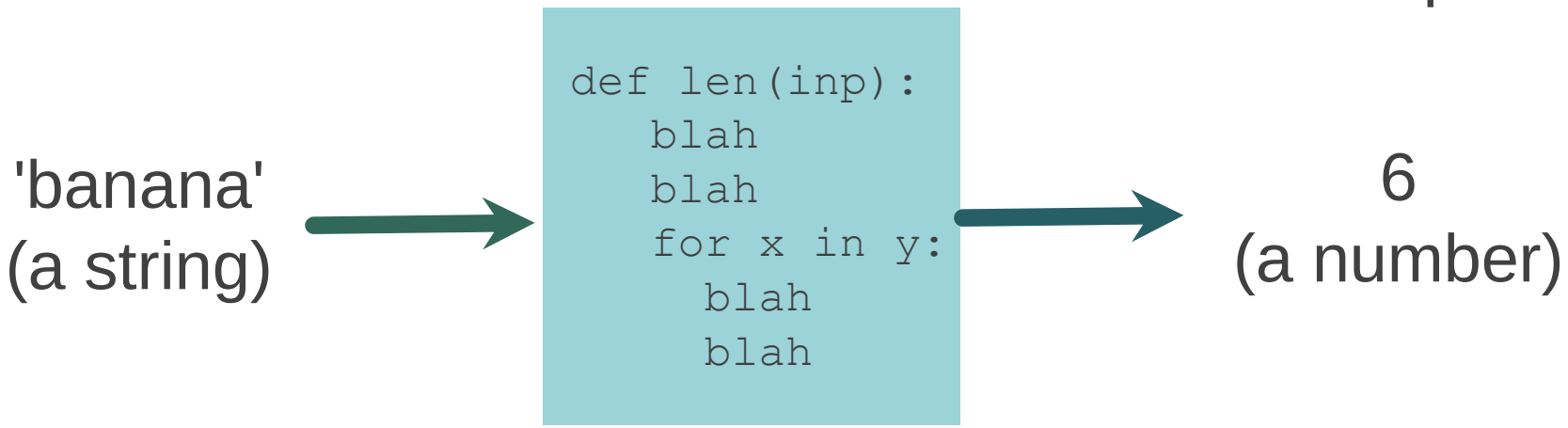
A function is some stored code that we use. A function takes some input and produces an output.



len Function

```
>>> fruit = 'banana'
>>> x = len(fruit)
>>> print(x)
6
```

A function is some stored code that we use. A function takes some input and produces an output.



Looping Through Strings

- Using a while statement, an iteration variable, and the len function, we can construct a loop to look at each of the letters in a string individually

```
fruit = 'banana'
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(index, letter)
    index = index + 1
```

```
0 b
1 a
2 n
3 a
4 n
5 a
```

Looping Through Strings

- A definite loop using a for statement is much more elegant
- The iteration variable is completely taken care of by the for loop

```
fruit = 'banana'
for letter in fruit:
    print(letter)
```

```
b
a
n
a
n
a
```

Looping Through Strings

- A definite loop using a for statement is much more elegant
- The iteration variable is completely taken care of by the for loop

```
fruit = 'banana'
for letter in fruit :
    print(letter)
```

```
b
a
n
a
n
a
```

```
index = 0
while index < len(fruit) :
    letter = fruit[index]
    print(letter)
    index = index + 1
```

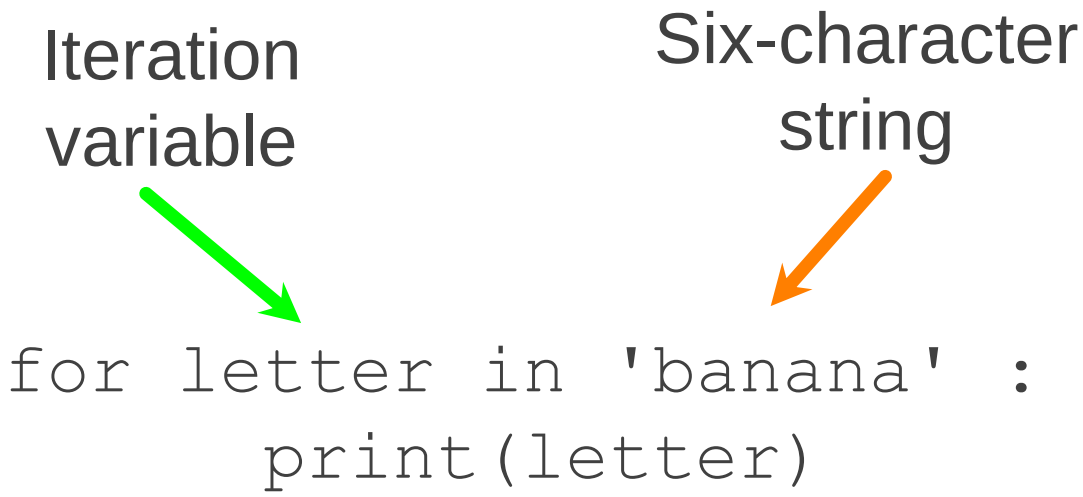
Looping and Counting

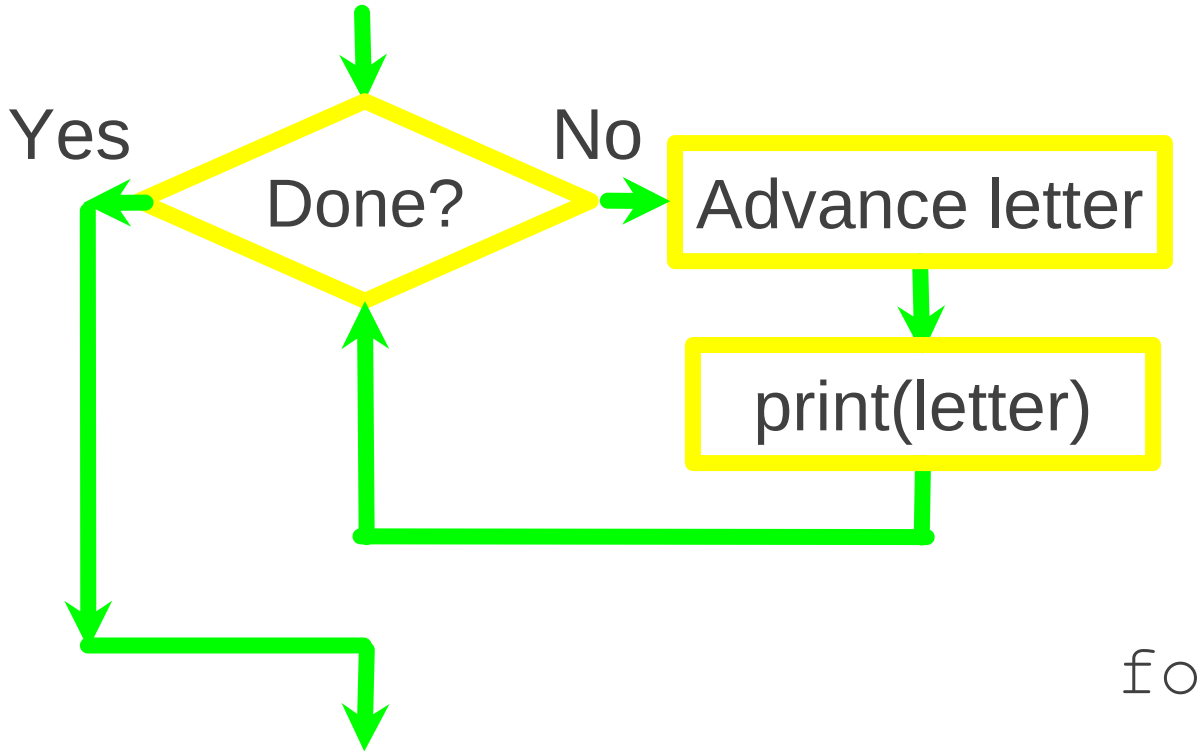
- This is a simple loop that loops through each letter in a string and counts the number of times the loop encounters the 'a' character

```
word = 'banana'
count = 0
for letter in word :
    if letter == 'a' :
        count = count + 1
print(count)
```

Looking Deeper into in

- The iteration variable “iterates” through the sequence (ordered set)
- The block (body) of code is executed once for each value in the sequence
- The iteration variable moves through all of the values in the sequence






```
for letter in 'banana' :
    print(letter)
```

The iteration variable “iterates” through the string and the block (body) of code is executed once for each value in the sequence

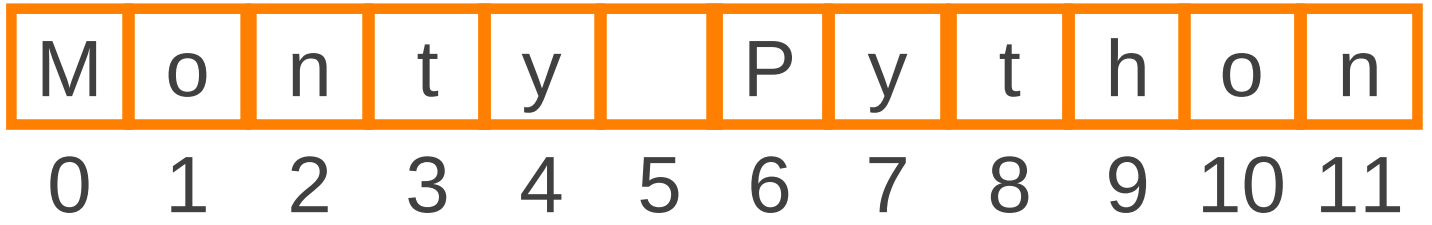
Advanced String Operations

ACTIVITY

Useful string functions & methods

Name	Purpose
`len(s)`	Calculate the length of the string `s`
`+`	Add two strings together
`*`	Repeat a string
`s.find(x)`	Find the first position of `x` in the string `s`
`s.count(x)`	Count the number of times `x` is in the string `s`
`s.upper()` `s.lower()`	Return a new string that is all uppercase or lowercase
`s.replace(x, y)`	Return a new string that has replaced the substring `x` with the new substring `y`
`s.strip()`	Return a new string with whitespace stripped from the ends
`s.format()`	Format a string's contents

Slicing Strings



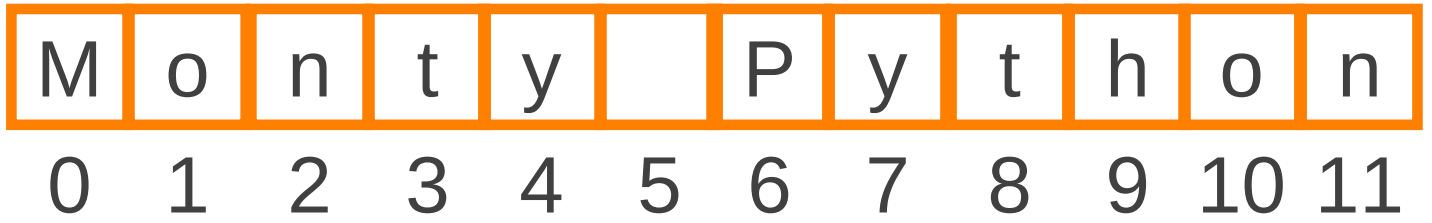
- We can also look at any continuous section of a string using a colon operator

  The second number is one beyond the end of the slice - “up to but not including”

- If the second number is beyond the end of the string, it stops at the end

```
>>> s = 'Monty Python'
>>> print(s[0:4])
Mont
>>> print(s[6:7])
P
>>> print(s[6:20])
Python
```

Slicing Strings



- If we leave off the first number or the last number of the slice, it is assumed to be the beginning or end of the string respectively

```
>>> s = 'Monty Python'
>>> print(s[:2])
Mo
>>> print(s[8:])
thon
>>> print(s[:])
Monty Python
```

String Concatenation

- When the + operator is applied to strings, it means "concatenation"

```
>>> a = 'Hello'
>>> b = a + 'There'
>>> print(b)
HelloThere
>>> c = a + ' ' + 'There'
>>> print(c)
Hello There
>>>
```

Using in as a Logical Operator

- The in keyword can also be used to check to see if one string is "in" another string
- The in expression is a logical expression that returns True or False and can be used in an if statement

```
>>> fruit = 'banana'
>>> 'n' in fruit
True
>>> 'm' in fruit
False
>>> 'nan' in fruit
True
>>> if 'a' in fruit :
...     print('Found it!')
...
Found it!
>>>
```

String Comparison

```
if word == 'banana':
    print('All right, bananas.')

if word < 'banana':
    print('Your word,' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word,' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

String Library

- Python has a number of string functions which are in the string library
- These functions are already built into every string - we invoke them by appending the function to the string variable
- These functions do not modify the original string, instead they return a new string that has been altered

```
>>> greet = 'Hello Bob'
>>> zap = greet.lower()
>>> print(zap)
hello bob
>>> print(greet)
Hello Bob
>>> print('Hi There'.lower())
hi there
>>>
```

```
>>> stuff = 'Hello world'
>>> type(stuff)
<class 'str'>
>>> dir(stuff)
['capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
```

https://docs.python.org/3/library/stdtypes.html#string-methods

`str.`**`replace`**(*old*, *new*[, *count*])

Return a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

`str.`**`rfind`**(*sub*[, *start*[, *end*]])

Return the highest index in the string where substring *sub* is found, such that *sub* is contained within `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` on failure.

`str.`**`rindex`**(*sub*[, *start*[, *end*]])

Like `rfind()` but raises `ValueError` when the substring *sub* is not found.

`str.`**`rjust`**(*width*[, *fillchar*])

Return the string right justified in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

`str.`**`rpartition`**(*sep*)

Split the string at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

`str.`**`rsplit`**(*sep=None*, *maxsplit=-1*)

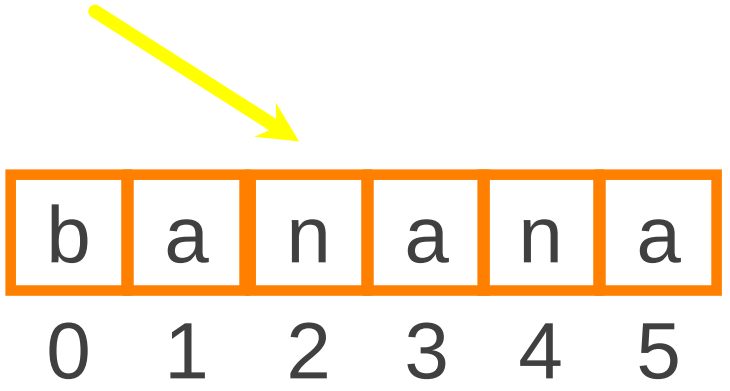
Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done, the *rightmost* ones. If *sep* is not specified or `None`, any whitespace string is a separator. Except for splitting from the right, `rsplit()` behaves like `split()` which is described in detail below.

String Library

```
str.capitalize()
str.center(width[, fillchar])
str.endswith(suffix[, start[, end]])
str.find(sub[, start[, end]])
str.lstrip([chars])
str.replace(old, new[, count])
str.lower()
str.rstrip([chars])
str.strip([chars])
str.upper()
```

Searching a String

- We use the find() function to search for a substring within another string
- find() finds the first occurrence of the substring
- If the substring is not found, find() returns -1
- Remember that string position starts at zero




```
>>> fruit = 'banana'
>>> pos = fruit.find('na')
>>> print(pos)
2
>>> aa = fruit.find('z')
>>> print(aa)
-1
```

Making everything UPPER CASE

- You can make a copy of a string in lower case or upper case
- Often when we are searching for a string using find() we first convert the string to lower case so we can search a string regardless of case

```
>>> greet = 'Hello Bob'
>>> nnn = greet.upper()
>>> print(nnn)
HELLO BOB
>>> www = greet.lower()
>>> print(www)
hello bob
>>>
```

Search and Replace

- The replace() function is like a “search and replace” operation in a word processor
- It replaces all occurrences of the search string with the replacement string

```
>>> greet = 'Hello Bob'
>>> nstr = greet.replace('Bob','Jane')
>>> print(nstr)
Hello Jane
>>> nstr = greet.replace('o','X')
>>> print(nstr)
HellX BXb
>>>
```

Stripping Whitespace

- Sometimes we want to take a string and remove whitespace at the beginning and/or end
- lstrip() and rstrip() remove whitespace at the left or right
- strip() removes both beginning and ending whitespace

```
>>> greet = '   Hello Bob  '
>>> greet.lstrip()
'Hello Bob  '
>>> greet.rstrip()
'   Hello Bob'
>>> greet.strip()
'Hello Bob'
>>>
```

Prefixes

```
>>> line = 'Please have a nice day'
>>> line.startswith('Please')
True
>>> line.startswith('p')
False
```

Parsing and Extracting


```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print(atpos)
21
>>> sppos = data.find(' ',atpos)
>>> print(sppos)
31
>>> host = data[atpos+1 : sppos]
>>> print(host)
uct.ac.za
```

Two Kinds of Strings

```
Python 2.7.10
>>> x = '이광춘'
>>> type(x)
<type 'str'>
>>> x = u'이광춘'
>>> type(x)
<type 'unicode'>
>>>
```

```
Python 3.5.1
>>> x = '이광춘'
>>> type(x)
<class 'str'>
>>> x = u'이광춘'
>>> type(x)
<class 'str'>
>>>
```

In Python 3, all strings are Unicode

List

LECTURE 2

A List is a Kind of Collection

- A collection allows us to put many values in a single “variable”
- A collection is nice because we can carry all many values around in one convenient package.

```
friends = [ 'Joseph', 'Glenn', 'Sally' ]

carryon = [ 'socks', 'shirt', 'perfume' ]
```

List Constants

- List constants are surrounded by square brackets and the elements in the list are separated by commas
- A list element can be any Python object - even another list
- A list can be empty

```
>>> print([1, 24, 76])
[1, 24, 76]
>>> print(['red', 'yellow',
'blue'])
['red', 'yellow', 'blue']
>>> print(['red', 24, 98.6])
['red', 24, 98.6]
>>> print([ 1, [5, 6], 7])
[1, [5, 6], 7]
>>> print([])
[]
```

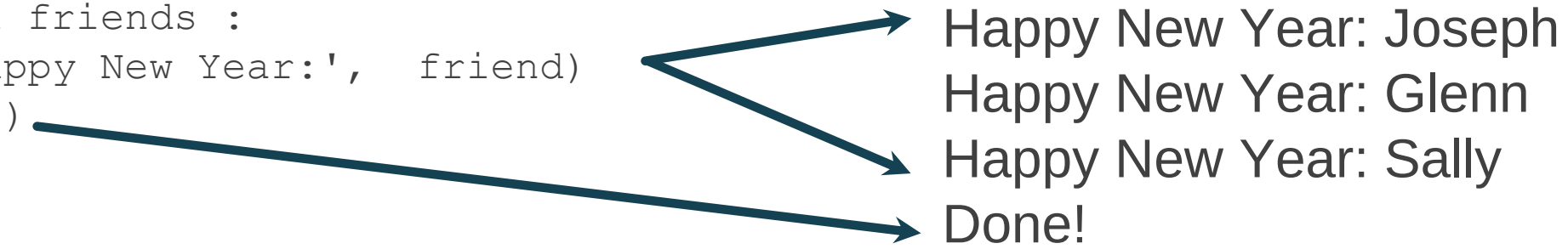
We Already Use Lists!

```
for i in [5, 4, 3, 2, 1] :
    print(i)
print('Blastoff!')
```

5
4
3
2
1
Blastoff!

Lists and Definite Loops - Best Pals

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends :
    print('Happy New Year:',  friend)
print('Done!')
```




```
z = ['Joseph', 'Glenn', 'Sally']
for x in z:
    print('Happy New Year:',  x)
print('Done!')
```

Looking Inside Lists

- Just like strings, we can get at any single element in a list using an index specified in square brackets


```
>>> friends = [ 'Joseph', 'Glenn', 'Sally' ]
>>> print(friends[1])
Glenn
>>>
```

Lists are Mutable

- Strings are “immutable” - we cannot change the contents of a string - we must make a new string to make any change
- Lists are “mutable” - we can change an element of a list using the index operator

```
>>> fruit = 'Banana'
>>> fruit[0] = 'b'
Traceback
TypeError: 'str' object does not
support item assignment
>>> x = fruit.lower()
>>> print(x)
banana
>>> lotto = [2, 14, 26, 41, 63]
>>> print(lotto)
[2, 14, 26, 41, 63]
>>> lotto[2] = 28
>>> print(lotto)
[2, 14, 28, 41, 63]
```

How Long is a List?

- The len() function takes a list as a parameter and returns the number of elements in the list
- Actually len() tells us the number of elements of any set or sequence (such as a string...)

```
>>> greet = 'Hello Bob'
>>> print(len(greet))
9
>>> x = [ 1, 2, 'joe', 99]
>>> print(len(x))
4
>>>
```

List Functions	Meanings
list.append(x)	Appends object x to list
list.count(x)	Returns count of how many times x occurs in list
list.remove(x)	Removes xect x from list
list.reverse()	Reverses objects of list in place
list.extend(seq)	Appends the contents of seq to list
list.index(x)	Returns the lowest index in list that x appears
list.insert(index, x)	Inserts xect x into list at offset index
list.pop(x=list[-1])	Removes and returns last object or x from list

Using the range Function

- The range function returns a list of numbers that range from zero to one less than the parameter
- We can construct an index loop using for and an integer iterator

```
>>> print(range(4))
[0, 1, 2, 3]
>>> friends = ['Joseph', 'Glenn', 'Sally']
>>> print(len(friends))
3
>>> print(range(len(friends)))
[0, 1, 2]
>>>
```

A Tale of Two Loops...

```
friends = ['Joseph', 'Glenn', 'Sally']

for friend in friends :
    print('Happy New Year:',  friend)

for i in range(len(friends)) :
    friend = friends[i]
    print('Happy New Year:',  friend)
```

```
>>> friends = ['Joseph', 'Glenn', 'Sally']
>>> print(len(friends))
3
>>> print(range(len(friends)))
[0, 1, 2]
>>>
```

Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally

Concatenating Lists Using +

- We can create a new list by adding two existing lists together

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
>>> print(a)
[1, 2, 3]
```

Lists Can Be Sliced Using :

```
>>> t = [9, 41, 12, 3, 74, 15]
>>> t[1:3]
[41,12]
>>> t[:4]
[9, 41, 12, 3]
>>> t[3:]
[3, 74, 15]
>>> t[:]
[9, 41, 12, 3, 74, 15]
```

Remember: Just like in strings, the second number is “up to but not including”

List Methods

```
>>> x = list()
>>> type(x)
<type 'list'>
>>> dir(x)
['append', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
>>>
```

http://docs.python.org/tutorial/datastructures.html

Building a List from Scratch

- We can create an empty list and then add elements using the append method
- The list stays in order and new elements are added at the end of the list

```
>>> stuff = list()
>>> stuff.append('book')
>>> stuff.append(99)
>>> print(stuff)
['book', 99]
>>> stuff.append('cookie')
>>> print(stuff)
['book', 99, 'cookie']
```

Is Something in a List?

- Python provides two operators that let you check if an item is in a list
- These are logical operators that return True or False
- They do not modify the list

```
>>> some = [1, 9, 21, 10, 16]
>>> 9 in some
True
>>> 15 in some
False
>>> 20 not in some
True
>>>
```

Lists are in Order

- A list can hold many items and keeps those items in the order until we do something to change the order
- A list can be sorted (i.e., change its order)
- The sort method (unlike in strings) means "sort yourself"

```
>>> friends = [ 'Joseph', 'Glenn', 'Sally' ]
>>> friends.sort()
>>> print(friends)
['Glenn', 'Joseph', 'Sally']
>>> print(friends[1])
Joseph
>>>
```

Built-in Functions and Lists

- There are a number of functions built into Python that take lists as parameters
- Remember the loops we built? These are much simpler.

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums)/len(nums))
25.6
```

Algorithm 1:

```
total = 0
count = 0
while True :
    inp = input('Enter a number: ')
    if inp == 'done' : break
    value = float(inp)
    total = total + value
    count = count + 1

average = total / count
print('Average:', average)
```

Algorithm 2:

```
numlist = list()
while True :
    inp = input('Enter a number: ')
    if inp == 'done' : break
    value = float(inp)
    numlist.append(value)

average = sum(numlist) / len(numlist)
print('Average:', average)
```

```
Enter a number: 3
Enter a number: 9
Enter a number: 5
Enter a number: done
Average: 5.66666666667
```

Best Friends: Strings and Lists

```
>>> abc = 'With three words'
>>> stuff = abc.split()
>>> print(stuff)
['With', 'three', 'words']
>>> print(len(stuff))
3
>>> print(stuff[0])
With
```

```
>>> print(stuff)
['With', 'three', 'words']
>>> for w in stuff :
...     print(w)
...
With
Three
Words
>>>
```

Split breaks a string into parts and produces a list of strings. We think of these as words. We can access a particular word or loop through all the words.

Example:

- When you do not specify a delimiter, multiple spaces are treated like one delimiter
- You can specify what delimiter character to use in the splitting

```
>>> line = 'A lot                    of spaces'
>>> etc = line.split()
>>> print(etc)
['A', 'lot', 'of', 'spaces']
>>>
>>> line = 'first;second;third'
>>> thing = line.split()
>>> print(thing)
['first;second;third']
>>> print(len(thing))
1
>>> thing = line.split(';')
>>> print(thing)
['first', 'second', 'third']
>>> print(len(thing))
3
>>>
```

Example:

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if not line.startswith('From ') : continue
    words = line.split()
    print(words[2])
```

Sat
Fri
Fri
Fri
...

```
>>> line = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
>>> words = line.split()
>>> print(words)
['From', 'stephen.marquard@uct.ac.za', 'Sat', 'Jan', '5', '09:14:16', '2008']
>>>
```

The Double Split Pattern

Sometimes we split a line one way, and then grab one of the pieces of the line and split that piece again

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008

words = line.split()
email = words[1]
print pieces[1]
```

The Double Split Pattern

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

```
words = line.split()
email = words[1]                          stephen.marquard@uct.ac.za
print pieces[1]
```

The Double Split Pattern

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

```
words = line.split()
email = words[1]
pieces = email.split('@')
print pieces[1]
```

```
stephen.marquard@uct.ac.za
['stephen.marquard', 'uct.ac.za']
```

The Double Split Pattern

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

```
words = line.split()
email = words[1]
pieces = email.split('@')
print(pieces[1])
```

```
stephen.marquard@uct.ac.za
['stephen.marquard', 'uct.ac.za']
'uct.ac.za'
```

Tuple

LECTURE 3

Tuples

Same as lists but

- Immutable
- Enclosed in parentheses
- A tuple with a single element ***must*** have a comma inside the parentheses:
  - **a = (11,)**

Examples

```
>>> mytuple = (11, 22, 33)
>>> mytuple[0]
11
>>> mytuple[-1]
33
>>> mytuple[0:1]
(11,)
```

The comma is required!

Why?

No confusion possible between **[11]** and **11**

(11) is a perfectly acceptable expression

- **(11) without the comma** is the integer 11
- **(11,) with the comma** is a list containing the integer 11

Sole dirty trick played on us by tuples!

Tuples are immutable

```
>>> mytuple = (11, 22, 33)
>>> saved = mytuple
>>> mytuple += (44,)
>>> mytuple
(11, 22, 33, 44)
>>> saved
(11, 22, 33)
```

Things that do not work

mytuple += 55

```
Traceback (most recent call last):Z
…
TypeError:
 can only concatenate tuple (not "int")
to tuple
```

- Can understand that!

Sorting tuples

```
>>> atuple = (33, 22, 11)
>>> atuple.sort()
Traceback (most recent call last):
…
AttributeError:
'tuple' object has no attribute 'sort'
>>> atuple = sorted(atuple)
>>> atuple
[11, 22, 33]
```

Tuples are immutable!

sorted( ) returns a list!

Most other things work!

```
>>> atuple = (11, 22, 33)

>>> len(atuple)
3

>>> 44 in atuple
False

>>> [ i for [i for i in atuple]
[11, 22, 33]
```

The reverse does not work

```
>>> alist = [11, 22, 33]

>>> (i for i in alist)
<generator object <genexpr> at 0x02855DA0>
```

Does not work!

Converting sequences into tuples

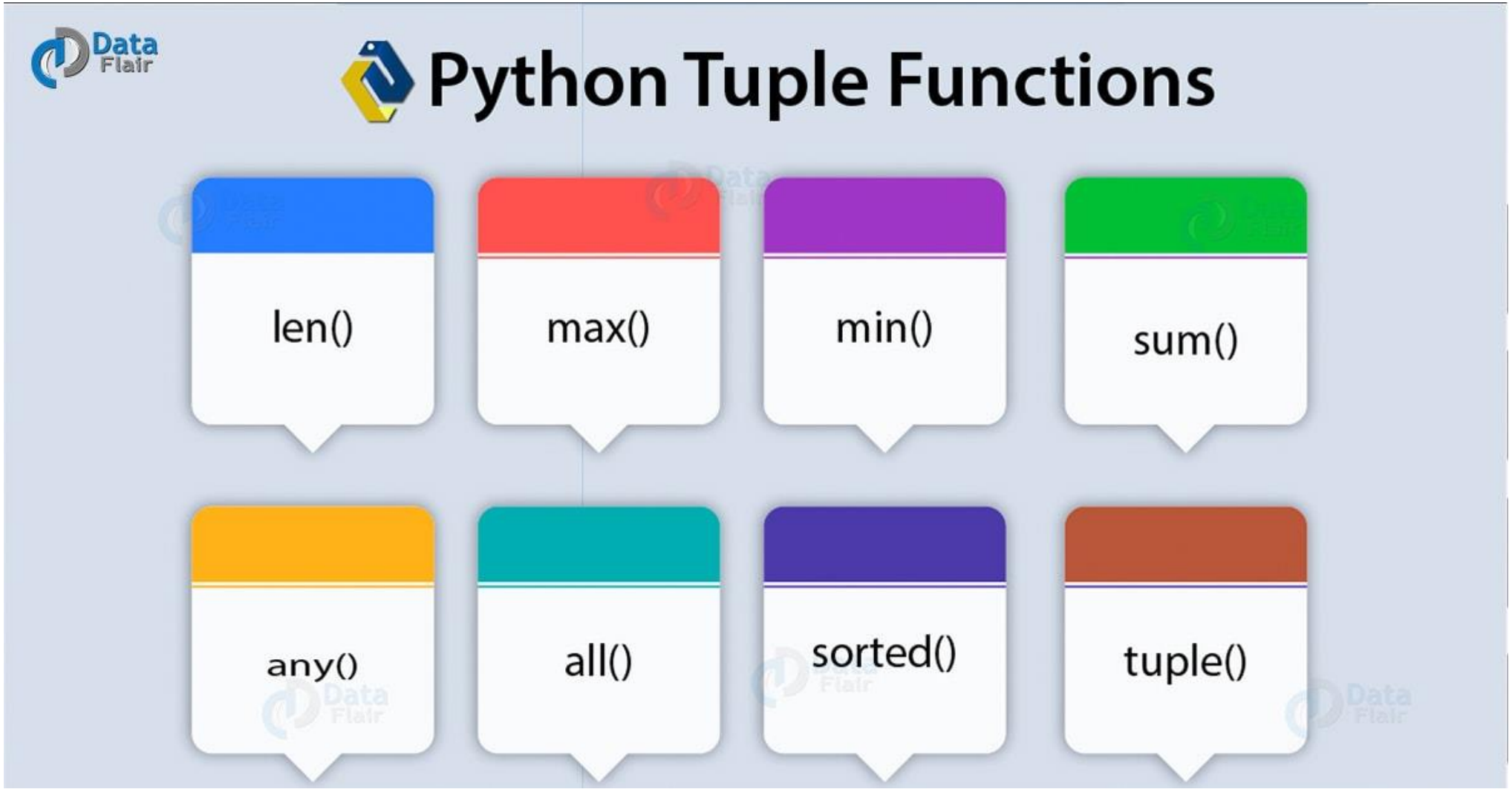
```
>>> alist = [11, 22, 33]

>>> atuple = tuple(alist)

>>> atuple
(11, 22, 33)

>>> newtuple = tuple('Hello World!')

>>> newtuple
('H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!')
```


Data Flair
Python Tuple Functions
len()
max()
min()
sum()
any()
all()
sorted()
tuple()

Set

LECTURE 4

Sets

Indentified by ***curly braces***

- **{'Alice', 'Bob', 'Carol'}**
- **{'Dean'}** is a ***singleton***

Can only contain ***unique elements***

- ***Duplicates are eliminated***

Immutable like tuples and strings

Sets do not contain duplicates

```
>>> cset = {11, 11, 22}

>>> cset
{11, 22}
```

Sets are immutable

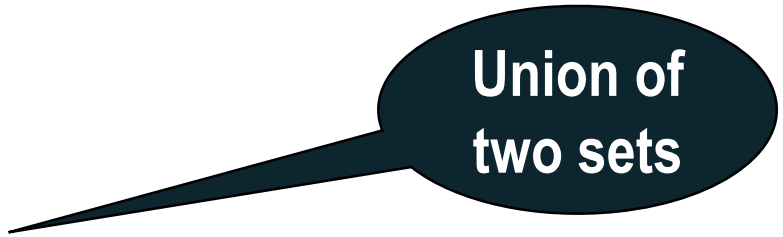
```
>>> aset = {11, 22, 33}

>>> bset = aset

>>> aset = aset | {55}

>>> aset
{33, 11, 22, 55}

>>> bset
{33, 11, 22}
```



Sets have no order

```
>>> {1, 2, 3, 4, 5, 6, 7}
{1, 2, 3, 4, 5, 6, 7}

>>> {11, 22, 33}
{33, 11, 22}
```

Sets do not support indexing

```
>>> myset = {'Apples', 'Bananas', 'Oranges'}

>>> myset
{'Bananas', 'Oranges', 'Apples'}

>>> myset[0]
Traceback (most recent call last):
   File "<pyshell#2>", line 1, in <module>
      myset[0]
TypeError: 'set' object does not support
indexing
```

Examples

```
>>> alist = [11, 22, 33, 22, 44]

>>> aset = set(alist)

>>> aset
{33, 11, 44, 22}

>>> aset = aset + {55}
SyntaxError: invalid syntax
```

Operation	Equivalent	Result
`len(s)`		number of elements in set *s* (cardinality)
`x in s`		test *x* for membership in *s*
`x not in s`		test *x* for non-membership in *s*
`s.issubset(t)`	`s <= t`	test whether every element in *s* is in *t*
`s.issuperset(t)`	`s >= t`	test whether every element in *t* is in *s*
`s.union(t)`	`s | t`	new set with elements from both *s* and *t*
`s.intersection(t)`	`s & t`	new set with elements common to *s* and *t*
`s.difference(t)`	`s - t`	new set with elements in *s* but not in *t*
`s.symmetric_difference(t)`	`s ^ t`	new set with elements in either *s* or *t* but not both
`s.copy()`		new set with a shallow copy of *s*

Boolean operations on sets (I)

Union of two sets

Contains all elements that are in set **A** **or** in set **B**

Boolean operations on sets (II)

Intersection of two sets

Contains all elements that are in both sets **A and B**

Boolean operations on sets (III)

Difference of two sets

Contains all elements that are **in A** but **not in B**

Boolean operations on sets (IV)

Symmetric difference of two sets

Contains all elements that are **either**

- **in** set **A** but **not in** set **B or**
- **in** set **B** but **not in** set **A**

Boolean operations on sets (V)

>>> aset = {11, 22, 33}

>>> bset = {12, 23, 33}

Union of two sets

- >>> aset | bset
  {33, 22, 23, 11, 12}

Intersection of two sets:

- >>> aset & bset
  {33}

Boolean operations on sets (VI)

>>> aset = {11, 22, 33}

>>> bset = {12, 23, 33}

Difference:

- >>> aset – bset
  {11, 22}

Symmetric difference:

- >>> aset ^ bset
  {11, 12, 22, 23}

Dictionary

LECTURE 5

Dictionaries (I)

Store ***pairs*** of entries called ***items***
{ 'CS' : '743-713-3350', 'UHPD' : '713-743-3333'}

Each pair of entries contains

- A ***key***
- ***A value***

Key and values are separated by a colon

Paris of entries are separated by commas

Dictionary is enclosed within curly braces

Usage

Keys must be ***unique*** within a dictionary

- No ***duplicates***

If we have

age = {'Alice' : 25, 'Bob' :28}

then

age['Alice'] is **25**

and

age[Bob'] is **28**

Dictionaries are mutable

```
>>> age = {'Alice' : 25, 'Bob' : 28}

>>> saved = age

>>> age['Bob'] = 29

>>> age
{'Bob': 29, 'Alice': 25}

>>> saved
{'Bob': 29, 'Alice': 25}
```

Keys must be unique

```
>>> age = {'Alice' : 25, 'Bob' : 28, 'Alice' : 26}

>>> age
{'Bob': 28, 'Alice': 26}
```

Displaying contents

```
>>> age = {'Alice' : 25, 'Carol': 'twenty-two'}

>>> age.items()
dict_items([ ('Alice', 25), ('Carol', 'twenty-two')])

>>> age.keys()
dict_keys([ 'Alice', 'Carol'])

 age.values()
dict_values([28, 25, 'twenty-two'])
```

Updating directories

```
>>> age = {'Alice': 26 , 'Carol' : 22}

>>> age.update({'Bob' : 29})

>>> age
{'Bob': 29, 'Carol': 22, 'Alice': 26}

>>> age.update({'Carol' : 23})

>>> age
{'Bob': 29, 'Carol': 23, 'Alice': 26}
```

Returning a value

```
>>> age = {'Bob': 29, 'Carol': 23, 'Alice': 26}

>>> age.get('Bob')
29

>>> age['Bob']
29
```

Removing a specific item (I)

```
>>> a = {'Alice' : 26, 'Carol' : 'twenty-two'}

>>> a
{'Carol': 'twenty-two', 'Alice': 26}

>>> a.pop('Carol’)
'twenty-two'

>>> a
{'Alice': 26}
```

Removing a specific item (II)

```
>>> a.pop('Alice')
26

>>> a
{}

>>>
```

Remove a random item

```
>>> age = {'Bob': 29, 'Carol': 23, 'Alice': 26}
>>> age.popitem()
('Bob', 29)
>>> age
{'Carol': 23, 'Alice': 26}
>>> age.popitem()
('Carol', 23)
>>> age
{'Alice': 26}
```

Summary

Strings, lists, tuples, sets and dictionaries all deal with aggregates

Two big differences

- ***Lists*** and ***dictionaries*** are ***mutable***
  - Unlike strings, tuples and sets
- ***Strings***, ***lists*** and ***tuples*** are ***ordered***
  - Unlike sets and dictionaries

Mutable aggregates

Can modify individual items

- **x = [11, 22, 33]**
  x[0] = 44
  will work

Cannot save current value

- **x = [11,22, 33]**
  y = x
  will **not** work

Immutable aggregates

Cannot modify individual items

- **s = 'hello!'**
  s[0] = 'H'
  is an **ERROR**

Can save current value

- **s= 'hello!'**
  t = s
  will work

Ordered aggregates

Entities in the collection can be accessed through a **numerical index**

- **s= 'Hello!'**
  s[0]
- **x = ['Alice', 'Bob', 'Carol']**
  x[-1]
- **t = (11, 22)**
  t[1]

Other aggregates

Cannot index sets

- **myset = {'Apples', 'Bananas', 'Oranges'} myset[0] is WRONG**

Can only index dictionaries **through their keys**

- **age = {'Bob': 29, 'Carol': 23, 'Alice': 26}**
  age['Alice'] works
  age[0] is **WRONG**

Python Dictionary Methods	
Method	**Description**
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and values
get()	Returns the value of the specified key
items()	Returns a list containing the a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

JSON	Python
object	dict
array	list
string	unicode
number (int)	int, long
number (real)	float
TRUE	TRUE
FALSE	FALSE
null	None