Python Programming

IOE 373 Lecture 14

Basic Elements

- Functions
- Loops and Iteration
- Strings

Functions

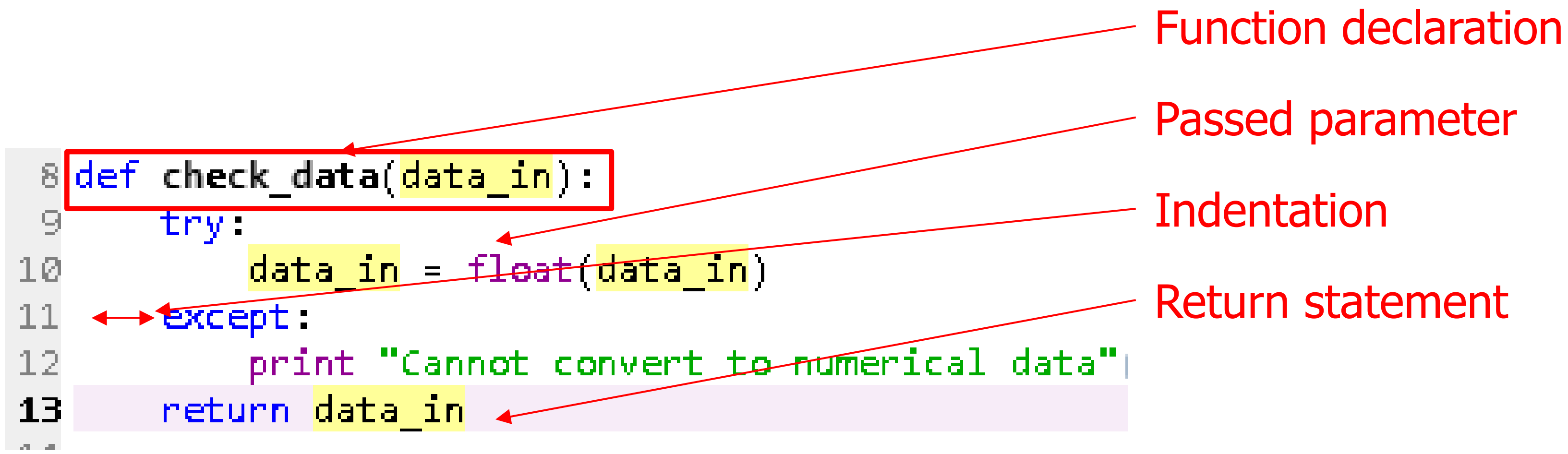
- Built-in Functions or Custom Functions
- For list of built-in Python functions:
  - https://docs.python.org/3/library/functions.html
- Some examples:
  - max(), min(), str(), int(), float()
  - range(), len(), abs(), pow()
  - sum()
  - open()

Custom Functions

- In Python a function is some reusable code that takes arguments(s) as input, does some computation, and then returns a result or results
- We define a function using the def reserved word
- We call/invoke the function by using the function name, parentheses, and arguments in an expression

Custom Functions

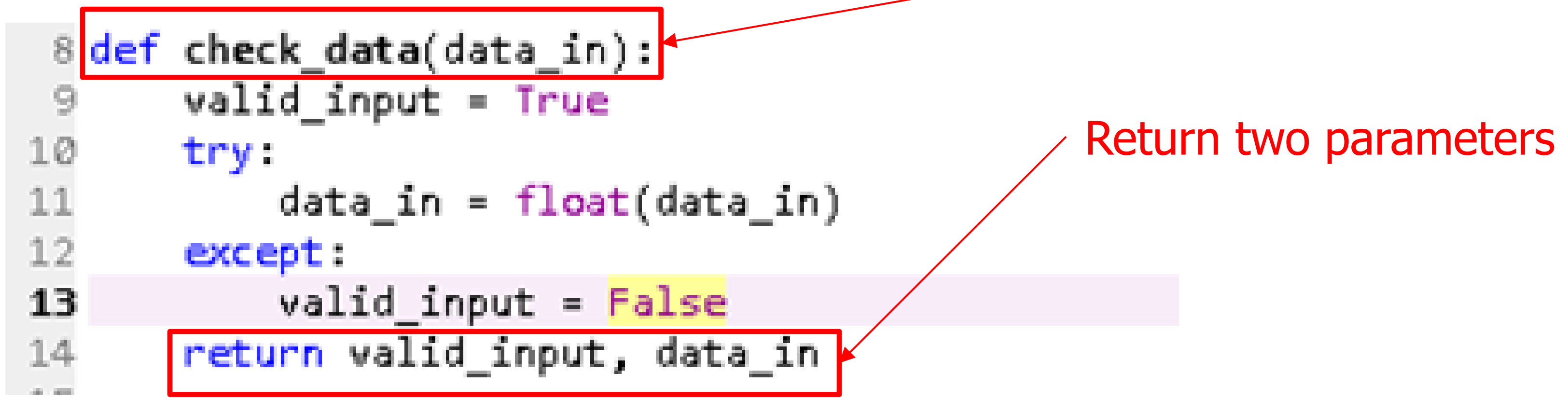
- A custom function to convert string data to float data



```
def check_data(data_in):
    try:
        data_in = float(data_in)
    except:
        print "Cannot convert to numerical data"
    return data_in
```

Custom Functions

- A further improvement



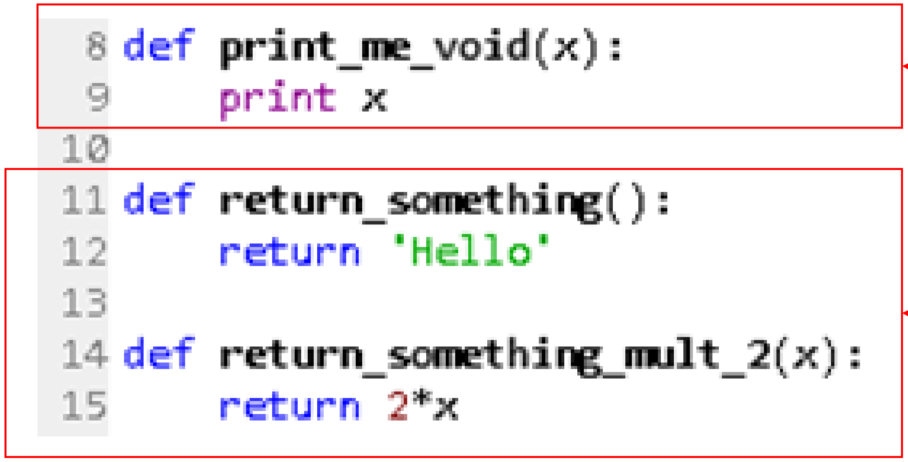
```
def check_data(data_in):
    valid_input = True
    try:
        data_in = float(data_in)
    except:
        valid_input = False
    return valid_input, data_in
```

Custom Functions

- Benefits of functions:
  - Only one instance of code to maintain
    - Less time to revise
    - Fewer errors because you don't an instance
  - Clearer code, more understandable
  - Shorter code
  - Provides for code reuse
    - Saves time writing code later
    - Functions are easily reused
    - … digging lines here and there out of a program takes longer and is fraught with error

Back to Functions

- Some functions return values, some do not
  - "When a function does not return a value, we call it a "void" function
  - Functions that return values are "fruitful" functions
  - Void functions are "not fruitful"



```
def print_me_void(x):
    print x

def return_something():
    return 'Hello'

def return_something_mult_2(x):
    return 2*x
```

"Void" or "Not Fruitful"

"Fruitful"

Loops

- Loops are a key tool in python
- Two scenarios. Execute a code block:
  - For a known number of times
    - Use a ‘for’ loop
  - For an unknown number of times
    - Until some criterion is satisfied
    - Use a ‘while’ loop

Loops

- for-loop Structure
  - Known number of iterations through loop

```
for i in [0, 1, 2]:
    print (i)
```

Performs the indented steps 3 times with, sequentially, i=0, i=1, i=2

- [0, 1, 2] is a list
  - Lists in Python are used for a variety of applications

Loops

- Built in range(x)function
  - This function creates a 'list' of integers based on the value x
  - x must be an integer
  - The list will contain x integers 0 through x–1: [0,…,x-1]
- This code does the same as the previous code:

```
for i in range(3):
     print (i)
```

Lists

- Another version of range() function
  - range(i,j) creates a list with integer elements starting at i and ending with j-1

Loops

- Still another version of range()
  - range(start,stop[,step])
  - Creates a ‘list’ of numbers which:
    - Starts at the value start
      - Default start = 0
    - step is optional argument
      - Default step is 1
    - Ends at the greatest possible value such that start + n×step that is less than stop
  - https://docs.python.org/3/library/functions.html#range
  - start/stop/step must be integers

Loops

- Typical loop code

Loop keeps iterating as long as this Boolean statement is `True`

```
keep_going = True
i = 0
while keep_going == True:
    i = i + 1
    if i >= 19:
        keep_going = False
```

Loops

- Loop control
  - **break** breaks out from the lowest level for or while loop
  - **continue** skips remainder of the current loop iteration and proceeds with the next iteration

Breaking Out of a Loop

- The break statement ends the current loop and jumps to the statement immediately following the loop
- It is like a loop test that can happen anywhere in the body of the loop

```
while True:
    line = input('> ')
    if line == 'done' :
        break
    print (line)
print ('Done!‛)
```

```
> hello there
hello there
> finished
finished
> done
Done!
```

Finishing an Iteration with continue

- The continue statement ends the current iteration and jumps to the top of the loop and starts the next iteration

```
while True:
    line = input('> ')
    if line[0] == '#' :
        continue
    if line == 'done' :
        break
    print (line)
print ('Done!')
```

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

Search Using a Boolean Variable

```
found = False
print ('Before', found)
for value in [9, 41, 12, 3, 74, 15] :
   if value == 3 :
         found = True
   print (found, value)
print ('After', found)
```

```
$ python search1.py
Before False
False 9
False 41
False 12
True 3
True 74
True 15
After True
```

If we just want to search and know if a value was found, we use a variable that starts at False and is set to True as soon as we find what we are looking for.

The “is” and “is not” Operators

```
smallest = None
print ('Before')
for value in [3, 41, 12, 9, 74, 15] :
    if smallest is None :
        smallest = value
    elif value < smallest :
        smallest = value
    print (smallest, value)
print ('After', smallest)
```

- Python has an **is** operator that can be used in logical expressions
- Implies “**is the same as**”
- **is not** also is a logical operator

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()
- We can convert numbers into a string using str()

```
>>> 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
>>> print (str(x))
‘124'
```

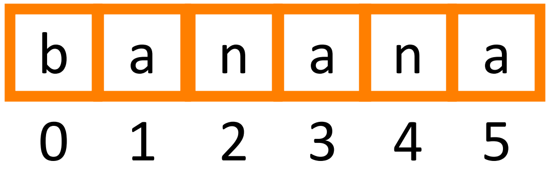
Reading and Converting

- 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
- Raw 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
```

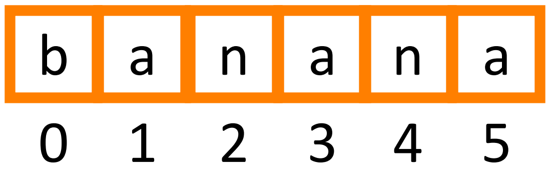
A Character Too Far

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

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

len Function

- There is a built-in function len that gives us the length of a string



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

Looping Through Strings

- Using a while statement and 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
```

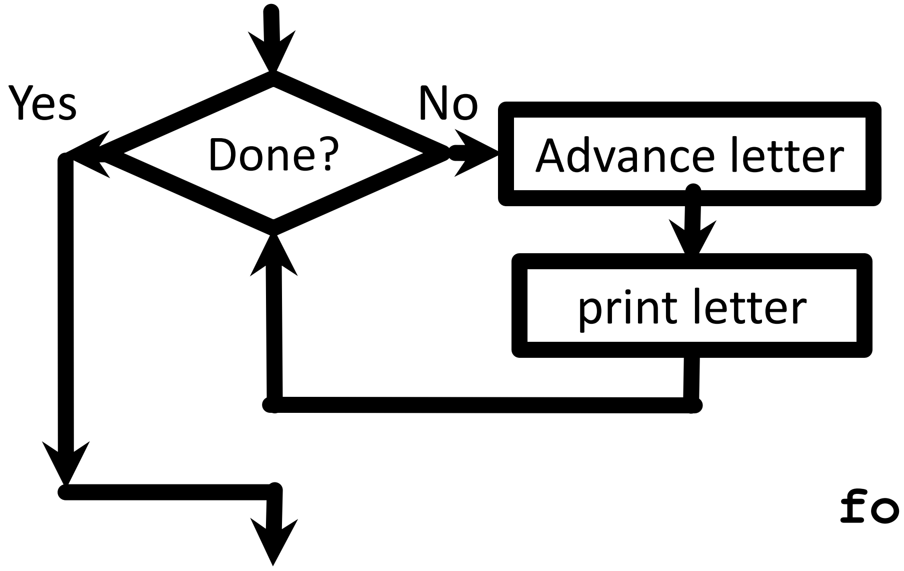
```
0 b
1 a
2 n
3 a
4 n
5 a
```

Looping Through Strings

- A definite loop using a for statement is similar to the VBA For-Each construct (without the need to declare variables)
- 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
```



```
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

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)
```

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

M	o	n	t	y		P	y	t	h	o	n
0	1	2	3	4	5	6	7	8	9	10	11

```
>>> 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

M	o	n	t	y		P	y	t	h	o	n
0	1	2	3	4	5	6	7	8	9	10	11

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

Slicing Strings - Reversing

- We can reverse strings using slicing

M	o	n	t	y		P	y	t	h	o	n
0	1	2	3	4	5	6	7	8	9	10	11

```
>>> s = 'Monty Python'
>>> print (s[::-1])
nohtyP ytnoM
```

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 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.')
```

Note: Python compares string lexicographically i.e using ASCII value of the characters.

Suppose you have str1 as "Mary" and str2 as "Mac" .

The first two characters from str1 and str2 (M and M) are compared.
As they are equal, the second two characters are compared.
Because they are also equal, the third two characters (r and c) are compared.
And because 'r' has greater ASCII value than 'c' , str1 is greater than str2

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
>>>
```

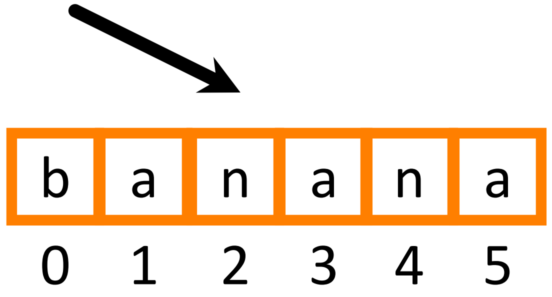
String Methods

```
>>> stuff = 'Hello world'
>>> type(stuff)
<type 'str'>
>>> dir(stuff)
['capitalize', 'center', 'count', 'decode', 'encode', 
'endswith', 'expandtabs', 'find', 'format', 'index', 
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 
'istitle', 'isupper', 'join', 'ljust', 'lower', 
'lstrip', '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

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 - startswith

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

Parsing and Extracting

```
                    21        31
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
```