Columbia Business School
AT THE VERY CENTER OF BUSINESS

Introduction to Python

Session 5

Recall: Data Structures

You've learned all of them!

Dictionary (Dict)

List (List)
Tuple (Tuple)

Boolean (Bool)
String (Str)
Integer (Int)
Float (Float)

Finding the Type of a Variable

```
type(variable_name)
```

What now?

What if you wanted to repeat a code block, but with a different input each time?

```
print(len(list1))
```

```
print(len(list2))
```

```
print(len(list3))
```

We’ve learned the use of For Loops or While Loops for repetition so far - Would they work?

Functions

Functions to the rescue

A function is a block of reusable code that can be used over and over easily.

A function is called on a given input, upon which it runs its code body on the input.

Example of a function call to do what we wanted just now:

```
length_printer(list_input)
```

We haven't written this function yet, but you can assume it does what its name states: It takes a list as an input and prints the length of this list

Functions to the rescue

How can we use the *length_printer* function to do what we need?
Call the function repeatedly on the multiple lists (list1, list2, list3) that we want to find the lengths of!

```
length_printer(list1)
length_printer(list2)
length_printer(list3)
```

Functions to the rescue

How does this save time?

Imagine if what you needed to do wasn’t as simple as simply printing the length of something.

Instead of rewriting the whole code block of things you need to do each time you want to run it on a new input, just type 1 line of code to call the function, which does everything for you on each new input!

Defining Functions = Write Your Own Functions

What you need to define:

- Your function name
  - Give it any name you want, ideally self-explanatory as good practice

- The inputs that your function will take:
  - Give them any names you want. Input names used at definition are only placeholders (more on this later!)

- The function body = What does your function do?

Defining Functions

stands for: define

Syntax

```
def function_name(input1, input2, …):
    # do something with inputs
    return ____
```

don't forget the colon!

Make sure all lines in the body of the function are indented!

Example from before:

```
def length_printer(lis):
    return len(lis)
```

Something new!

Tell Python what to do with the input *lis*

Defining Functions

Syntax

```
def function_name(input1, input2, …):
  # do something with inputs
  return ____
```

Inputs to functions can be:

- Any variable type we've learned so far (strings, numbers, booleans, lists, tuples, dictionaries)

There can be unlimited number of inputs to a function

Defining Functions

Syntax

```
def function_name(input1, input2, …):
  # do something with inputs
  return ____
```

Tell Python what your function must return (= output)

- Typically written as the last line of the function body
- A function can have more than 1 outputs
- *return* is an important concept! We will cover more on it later:
  - Difference between *return* and *print()*
  - When to use *return* and when to use *print()*

The difference between *return* and *print()* is a common pitfall for Python beginners!

Defining Functions: Example With Multiple Inputs

Given a dictionary of characters on The Office and their romantic partners, write a function that checks whether a character has a romantic partner on the show or not. If yes, return the name of their partner. Otherwise, return None.

```
romance_on_the_office = {'Jim': 'Pam', 'Michael':
'Holly', 'Dwight': 'Angela', 'Bob Vance': 'Phyllis'}

def check_romantic_partners(dic, name):
   return dic.get(name)
```

Recall: *dictionary.get(key)* is a dictionary method that returns the value associated with the key if the key exists in the dictionary, and None if it does not

Calling Functions
= Using a function after you've defined it

Syntax

```
function_name(var1, var2, …)
```

Use the variables you actually want to run the function on

This can be different from the input names you used when defining your function

- Recall on Slide 8: "Input names used at definition are only placeholders"

Confused? Let's look at an example!

Calling Functions

Recall the function named *check_romantic_partners()* we defined just now:

```
def check_romantic_partners(dic, name):
  return dic.get(name)
```

From the above definition, our function *check_romantic_partners()* must take 2 inputs, which we named *dic* and *name*. These names are placeholders for the actual variables we want to use the function on later.

Calling Functions

Recall we wanted to use *check_romantic_partners()* on the following dictionary:

```
romance_on_the_office = {'Jim': 'Pam', 'Michael':
'Holly', 'Dwight': 'Angela', 'Bob Vance': 'Phyllis'}
```

This dictionary will therefore be passed in as the first input to *check_romantic_partners():*

```
check_romantic_partners(romance_on_the_office, _____)
```

We still need another input to *check_romantic_partners()*, because we defined it to take 2 inputs. From our definition, this 2nd input is the name of a character, and we want to check whether he/she has a romantic partner on the show or not.

Calling Functions

```
romance_on_the_office = {'Jim': 'Pam', 'Michael':
'Holly', 'Dwight': 'Angela', 'Bob Vance': 'Phyllis'}
```

Say we want to check for the character Jim, so we call the function as follows:

```
print(check_romantic_partners(romance_on_the_office, 'Jim'))
```

What will this function call return?

```
'Pam'
```

Calling Functions

```
romance_on_the_office = {'Jim': 'Pam', 'Michael':
'Holly', 'Dwight': 'Angela', 'Bob Vance': 'Phyllis'}
```

Say we want to check for the character Creed, so we call the function as follows:

```
print(check_romantic_partners(romance_on_the_office, 'Creed'))
```

What will this function call return?

```
None
```

Calling Functions

Function Definition

```
def check_romantic_partners(dic, name):
  return dic.get(name)
```

Example of Function Call

```
check_romantic_partners(romance_on_the_office, 'Creed')
```

The sequence in which input variables are passed into a function call must tally with the positions of input name placeholders.

Here, `dic=romance_on_the_office` and `name='Creed'`

Calling Functions

What if we passed in input variables the other way round?

Function Definition

```
def check_romantic_partners(dic, name):
  return dic.get(name)
```

Example of WRONG Function Call

```
check_romantic_partners('Creed', romance_on_the_office)
```

Now, `dic= 'Creed'` and `name=romance_on_the_office`

Calling Functions

What if we passed in input variables the other way round?

Function Definition

```
def check_romantic_partners(dic, name):
  return dic.get(name)
```

Python will try to do this: `'Creed'.get(romance_on_the_office)`
which will throw this error:

```
AttributeError: 'str' object has no attribute 'get'
```

What's wrong? `'Creed'` is a string and `.get()` is not a valid string method!

Functions can be called on different inputs

Say you have another dictionary of romantic partners on a different sitcom, and you also want to check whether a character has a romantic partner on that show or not. Similarly, if yes, return the name of their partner. Otherwise, return None.

```
romance_on_friends = {'Ross': 'Rachel', 'Chandler':
'Monica', 'Mike': 'Phoebe'}
```

You already have a function written to do this! So just pass your new inputs into the same function:

```
print(check_romantic_partners(romance_on_friends, 'Chandler'))
```

```
'Monica'
```

Function Concepts

- When defining functions, input names used are merely placeholders and can be replaced with the required input variable during a function call.

- When calling functions, the sequence of passing in input variables is important
  - The sequence in which input variables are passed into a function call must tally with the positions of input name placeholders.

Important backtrack: *return*

Recall: Syntax for function definition

```
def function_name(input1, input2, …):
  # do something with inputs
  return ____
```

What does *return* do?

The keyword *return* in a function body tells Python to give an output when the function is called.

Example:

```
romance_on_friends = {'Ross': 'Rachel', 'Chandler': 'Monica',
'Mike': 'Phoebe'}

chandlers_partner = check_romantic_partners(romance_on_friends, 'Chandler')
print(chandlers_partner)
```

```
'Monica'
```

The function call above gives the output `'Monica'`. This output is then assigned to the variable `chandlers_partner`.

What does *return* do?

So what? This is useful because we can subsequently use the function output in future lines of code, since it is now stored in the variable `chandlers_partner`.

Usefulness of *return*:

Allows us to store function outputs in variables for them to be used again later!

return versus *print()*

Could we have used *print()* instead of *return*?

The difference between *return* and *print()* is a common pitfall for Python beginners!

New syntax for function definition using print()

```
def function_name(input1, input2, …):
  # do something with inputs
  print(____)
```

Example of syntax usage

```
def check_romantic_partners(dic, name):
  print(dic.get(name))
```

return versus *print()*

Could we have used *print()* instead of *return*?

```
def check_romantic_partners(dic, name):
  print(dic.get(name))
```

What happens now when we call *check_romantic_partners()*?

```
romance_on_friends = {'Ross': 'Rachel', 'Chandler': 'Monica',
'Mike': 'Phoebe'}

check_romantic_partners(romance_on_friends, 'Chandler')
```

```
'Monica'
```

Python prints the output `'Monica'`

return versus *print()*

Could we have used *print()* instead of *return*?

```
def check_romantic_partners(dic, name):
  print(dic.get(name))
```

What if we call *check_romantic_partners()* and try to assign its output to a variable?

```
romance_on_friends = {'Ross': 'Rachel', 'Chandler': 'Monica',
'Mike': 'Phoebe'}

chandlers_partner = check_romantic_partners(romance_on_friends, 'Chandler')
```

```
'Monica'
```

The output `'Monica'` is printed even though we never used *print()*!

Because we defined our function to print `dic.get(name)` each time the function body is run.

So every time the function *check_romantic_partners()* is called, Python automatically prints the result without us having to use *print()*

return versus *print()*

Could we have used *print()* instead of *return*?

But what was assigned to the variable `chandlers_partner`? Let's see it by calling *print()* on it

```
romance_on_friends = {'Ross': 'Rachel', 'Chandler': 'Monica',
'Mike': 'Phoebe'}
```

```
chandlers_partner = check_romantic_partners(romance_on_friends, 'Chandler')
```

```
'Monica'
```

The output `'Monica'` is printed even though we never used *print()*!

```
print(chandlers_partner)
```

```
None
```

There is nothing assigned to `chandlers_partner` now!

Why? (Next slide)

return versus *print()*

Could we have used *print()* instead of *return*?

Recall: What return does

The keyword *return* in a function body tells Python to give an output when the function is called.

Without using *return* in our function body, calling the function will NOT give any output.

Recall from Session 1: What print() does

print() displays the value of a variable onto the screen. This value is only displayed, and nothing else can be done to it!

return versus *print()*

When to use *print()* versus *return*?

When to use return

Only used inside function definitions (Why not outside? More on this later)

Use when: You want to obtain outputs from a function

- You can store these outputs in variables for them to be used again later

When to use print()

Can be used both inside and outside of function definitions

Use when: You want to display things on your screen and do nothing more with these displays

- Difference from *return*: You cannot assign this displayed value to a variable!

return versus *print()*

Recall: Printing in Jupyter Notebook

Jupyter Notebook automatically prints the last line of the code.
If you want an output to be printed, best to explicitly call `print()`.

```
print('Hello World again')
'Hello World'
```

```
Hello World again
'Hello World'
```

return versus *print()*

Recall: Printing in Jupyter Notebook

Jupyter Notebook automatically prints the last line of the code.
If you want an output to be printed, best to explicitly call `print()`.

```
'Hello World'
print('Hello World again')
```

```
Hello World again
```

return versus *print()*

Word of Caution:
Majority of students who fail the waiver exam fail because they don’t know the difference between return and print(), and when to use each!

Removing *return* entirely

New syntax for function definition, without return

```
def function_name(input1, input2, …):
  # do something with inputs
```

Example of syntax usage

```
def check_romantic_partners(dic, name):
  dic.get(name)
```

Removing *return* entirely

```
def check_romantic_partners(dic, name):
  dic.get(name)
```

What happens now when we call *check_romantic_partners()*?

```
romance_on_friends = {'Ross': 'Rachel', 'Chandler': 'Monica',
'Mike': 'Phoebe'}

print(check_romantic_partners(romance_on_friends, 'Chandler'))
```

```
None
```

Without using *return* in our function body, calling the function will NOT give any output.

Removing *return* entirely

```
def check_romantic_partners(dic, name):
  dic.get(name)
```

What if we call *check_romantic_partners()* and try to assign its output to a variable?

```
romance_on_friends = {'Ross': 'Rachel', 'Chandler': 'Monica',
'Mike': 'Phoebe'}

chandlers_partner = check_romantic_partners(romance_on_friends, 'Chandler')
print(chandlers_partner)
```

```
None
```

There is no output from calling the function, so there is no output assigned to the variable `chandlers_partner`!

Overall Concepts: Replacing *return*

1) **If *return* is replaced with *print()*:**

   Calling the function will merely display the function output to the screen

2) **If *return* is not replaced with anything and simply removed:**

   Nothing happens when calling the function

   - No function output
   - No displays on the screen

Recap: When to use *return*

The keyword *return* in a function body tells Python to give an output when the function is called.

- A function has an output ONLY if return is used in its function body

When to use *return*

Use *return* outside of function definitions?

return is a syntax only used in function definitions

Outside of function definitions, using *return* throws a syntax error:

Example:

```
while (n > 0):
    counter = counter * n
    n = n - 1
    return counter
```

```
SyntaxError: 'return' outside function
```

When to use *return*

Bottomline:

Use *return* only when defining functions. Otherwise, you don't need it!

Remember this, and you won't get confused

What happens after *return*?

The function body stops executing once the first *return* line is run

- The output is immediately given
- The rest of the function body is not run

Example:

```
def check_romantic_partners(dic, name, lis):
    return dic.get(name)
    lis.append(1)
    return lis
```

More than 1 return lines in this function definition

What happens after *return*?

```
def check_romantic_partners(dic, name, lis):
   return dic.get(name)
   lis.append(1)
   return lis

random_list = []
romance on friends = {'Ross': 'Rachel',
'Chandler': 'Monica', 'Mike': 'Phoebe'}

chandlers_partner =
check_romantic_partners(romance_on_friends, 'Chandler',
random_list)
```

What happens after *return*?

```
def check_romantic_partners(dic, name, lis):
    return dic.get(name)
    lis.append(1)
    return lis

print(chandlers_partner)
```

```
'Monica'
```

The function stops executing after hitting the first *return*, which is *return dic.get(name)*, and outputs *dic.get(name)*

```
print(random_list)
```

```
[]
```

The line *lis.append(1)* is never run since the function body stopped executing before getting to it, so nothing is appended into *random_list* and it remains empty

What happens after *return*?

What if we swapped the positions of the 2 return functions?

```
def check_romantic_partners(dic, name, lis):
  lis.append(1)
  return lis
  return dic.get(name)

chandlers_partner =
check_romantic_partners(romance_on_friends,
'Chandler', random_list)
```

What happens after *return*?

What if we swapped the positions of the 2 return functions?

```
def check_romantic_partners(dic, name, lis):
    lis.append(1)
    return lis
    return dic.get(name)

print(chandlers_partner)
```

```
[1]
```

The function stops executing after hitting the first *return*, which is *return lis*, and outputs *lis*

```
print(random_list)
```

```
[1]
```

The line *lis.append(1)* is now run before the function body stops executing, so 1 is appended into *random_list*

What happens after *return*?

The function body stops executing once the first *return* line is run

- The output is immediately given
- The rest of the function body is not run

Implications: When using if statements inside your function body!

Example:

```
def comparing_numbers(num1, num2):
   if num1 == num2:
      return num1
   if num1 ≤ num2:
      return num2
   if num2 < num1:
      return num1
```

What happens after *return*?

```
def comparing_numbers(num1, num2):
    if num1 == num2:
        return num1
    if num1 ≤ num2:
        return num2
    if num2 < num1:
        return num1

print(comparing_numbers(5, 5))
```

```
5
```

The 1st *if* statement (*if num1==num2*) evaluates to *True*, so Python runs *return num1* and the function body stops executing, even though the 2nd *if* statement (*if num1 ≤ num2*) is also *True*

Returning multiple outputs

Syntax

```
def function_name(input1, input2, …):
  # do something with inputs
  return ____, ____, ...
```

- Use commas to separate each output to be returned
- Returns a tuple with each output as an element within it

Returning multiple outputs

```
def check_romantic_partners(dic, name, lis):
    lis.append(1)
    return lis, dic.get(name)

random_list = []
romance_on_friends = {'Ross': 'Rachel',
'Chandler': 'Monica', 'Mike': 'Phoebe'}

outputs = check_romantic_partners(romance_on_friends,
'Chandler', random_list)

print(outputs)
```

```
([1], 'Monica')
```

Exercises

Try it! ★☆☆

You are an aspiring novelist striving to make your mark on the literary scene. While brainstorming potential ideas, you hit a creative roadblock. You choose to use your Python skills to assist you with part of the creative process. The following few exercises will test your ability to manipulate strings and in the process, finish your novel!

Try it! ★☆☆

Write a Python function, `both_ends`, that takes as input a string and returns the first 2 characters concatenated together with the last 2 characters. If the length of the string is less than 2, return the empty string instead.

For instance, 'bagel' yields 'bael' and 'so' yields 'soso'.

Solution

```
def both_ends(s):
    if len(s) < 2:
        return ""
    else:
        return s[:2] + s[-2:]
```

Try it! ★☆☆

Instead of the previous censoring approach, now you want to only reveal the first character of the word and replace the rest of the characters with '*'.

Write a function, `first_character`, to accomplish this.

For instance, 'bubble' yields 'b*****' and 'aardvark' yields 'a*******'.

Solution

```
def first_character(s):
    first_char = s[0]
    num_stars = len(s) - 1
    return first_char + num_stars * '*'
```

Try it! ★★☆

Now, you want to censor words in the following manner. Given a word, return a string where all occurrences of its first character have been changed to '*', except for the first character which should remain unchanged.

Write a function, `fix_start`, to accomplish this.

For instance, 'bubble' yields 'bu**le' and 'aardvark' yields 'a*rdv*rk'.

Solution

```
def fix_start(s):
    to_replace = s[0]
    back = s[1:]
    new_string = back.replace(str(to_replace), "*")
    return to_replace + new_string
```

Try it! ★☆☆

You want to use Python to automate some of your writing process. Specifically, you want to write a function, `simile_builder`, that takes as input a string and returns a simile built from the string.

The input should contain two words, separated by a colon ":", and your function should return a string in the form of 'a x is like a y' where x and y are the first and second words respectively. If the string does not contain a colon, return 'No simile, sad times.'

Try it!

For example,

`simile_builder('dog:best friend')` returns `'a dog is like a best friend'`

while

`simile_builder('bleh')` returns `'No simile, sad times.'`

Solution

```
def simile_builder (s):
    if ':' not in s:
        return 'No simile, sad times.'
    else:
        colon_index = s.index(':')
        first_term = s[:colon_index]
        second_term = s[colon_index + 1:]
        return 'a ' + first_term + ' is like a ' + second_term
```

Try it! ★★★

Write a function, `exclamation(s)`, that given a string s, look for the first exclamation mark. If there is a substring of 1 or more alphabetic characters immediately to the left of the exclamation mark, return this substring including the exclamation mark. Otherwise, return the empty string. You believe this helps in your writing by identifying the common words associated with the exclamation mark.

Try it!

For example,

`exclamation('Yippie yay! hurrah')` returns `'yay!'`

while

`exclamation('Nothing ! to shout about')` returns `''`

Solution

```
def exclamation(s):
    mark = s.find('!')
    if mark == -1:
        return ''
    i = mark - 1
    while i >= 0 and s[i].isalpha():
        i -= 1
    word = s[i + 1: mark + 1]
    if len(word) >= 2:
        return word
    return ''
```

Try it! ★★☆

You are going over your department's payroll when you realized that there has been a clerical error. Some of your employees' name appear more than once which will result in overpayment.

Given a list of names, write a function, `remove_duplicates`, that returns a list that contains only unique names. You need to ensure that your code is case insensitive.

```
names = ['Janice', 'Catherine', 'Karen', 'toby', 'Catherine',
'janice', 'barbara']
```

`remove_duplicates(names)` returns `['janice', 'catherine', 'karen', 'toby', 'barbara']`

Solution

```
def remove_duplicates(namelist):
    result = []
    for name in namelist:
        name = name.lower()
        if name not in result:
            result.append(name)
    return result
```

Try it! ★★★

In the old days long before social media was invented, people turned to phone books to find mutual friends. Your goal is to write a function, `mutual_friends`, that returns a new dictionary of mutual friends from two input dictionaries that contain the same key/value pair. The input dictionaries have names as keys and phone numbers as values.

Try it!

For example, the following two input dicts

```
{'Greg':6829, 'Jasmine':7777, 'Elijah':1005},
{'Greg':6829, 'Jasmine':7997, 'Rajesh':1005}
```

will return

```
{'Greg':6829}
```

Solution

```
def mutual_friends (phonebook_one, phonebook_two):
    mutual_friend = {}
    for name in phonebook_one:
        phone_num = phonebook_one[name]
        if name in phonebook_two and phonebook_two[name] == phone_num:
            mutual_friend[name] = phone_num
    return mutual_friend
```