PYTHON-Basics

Variables: used to store values

```
my_name = "Jane Doe"
print(my_name)
```

Data types:

Data Type	Description	Example
int	32 bit Integer	`25, 50`
long	Integer > 32 bits	`500L`
float	Floating point number	`29.99, 79.66`
bool	Boolean	`True, False`
str	Character sequence	`‘Python’`
tuple	Immutable sequence	`(2, 4, 6)`
list	Mutable sequence	`[‘Thinkful’, 5.5, 120]`
dict	Mapping keys and values	`{‘Cust_id’:1, ‘Name’: ‘John’}`

Functions: Named blocks of code, designed to do a specific job.

Information passed to a function is called an arguments, and information received by a function is called a parameters. Functions can return values.

A simple function	`def greeting():`
	`    print (‘Hello There’)`
	`greeting()`

Function with parameters	`def double(num):`
	`    print (num * 2)`
	`double(6)`

Function returning value	`def double(num):`
	`    return num * 2`
	`print(double(6))`

Working with String

```
first_name = 'Jane'
last_name = 'Doe'
full_name = first_name + ' ' + last_name
print(full_name)

Jane Doe
```

Escape characters: \n, \t

```
employees = 'FIRST\tLAST\nJohn\tCleese\nEric\tIdle'
print(employees)

FIRST   LAST
John    Cleese
Eric    Idle
```

String indexing and slicing

`str = ‘John Doe’`

`str[0]`	J
`str[0:4]`	John
`str[6:8]`	oe
`str[3:]`	n Doe
`str[:4]`	John

String Methods

Method	Returns
str.**capitalize()**	a string with first letter capitalized
str.**lower()**	lowercase string of a given string
str.**upper()**	uppercase string of a given string
str.**islower()**	True if all alphabets in a string are lowercase. False if any uppercase letter is present
str.**isupper()**	True if the string is all uppercase, otherwise False
str.**isdecimal()**	True if all characters in the string are decimal, otherwise False
str.**isalpha()**	True if all characters in the string are alphabets (can be both lowercase and uppercase). False if at least one character is not alphabet
str.**find(‘substring’)**	Integer index of the first occurrence of the substring. -1 if the substring is not found
str.**endswith('suffix')**	True if a string ends with the specified suffix, False otherwise.
str.**split('separator')**	breaks up a string at the specified separator and returns a list of strings. If the separator is not specified, any whitespace (space, newline etc.) is a separator.
str.**join(iterable)**	a string that is created by concatenating each element of an iterable.

Formatting strings

"Format" a string by replacing `{}` with the arguments you supply to the format function.

'{}, {}, {}'.**format**(0, 1, 2) -> ‘0,1,2’

```
'Let me have a {} with {} dashes of
{}'.format('whiskey', 3, 'bitters')

Let me have a whiskey with 3 dashes of bitters
```

Working with Numbers

Arithmetic operators

Operator	Example
+ (addition)	2 + 3.5 = 5.5
- (subtraction)	3 - 1 = 2
*** (multiplication)**	3.5 * 2 = 7.0
/ (true division)	5 / 2 = 2.5
// (floor division)	5 // 2 = 2
% (modulo)	5 % 2 = 1
**** (exponentiation)**	2 ** 3 = 8

Comparison operators

Operator	Example
< (less than)	4 < 5 = True
<= (less than or equal to)	1 <= 2 = True
> (greater than)	-5.2 > -7.5 = True
>= (greater than or equal)	18 >= 0 = True
== (equal)	1 == 1 = True
!= (not equal)	5 != '5' = True

Resources: Data Science prep course

| pub: 2018-04-23

Application Logic

Booleans and truthiness: Use bool() to find the truth status

Example	True/False
bool(true)	True
bool(false)	True or False = True
Numbers and strings evaluate to True (except 0 and empty string)	
bool(1)	True
bool(2)	True
bool(-1)	True
bool('Hello')	True
bool(' ')	True
0 and empty string evaluates to false	
bool(0)	False
bool('')	False
Collections evaluate to True	
bool([1, 2, 3])	True
bool({'arms': 2, 'sword': None})	True
empty collections evaluates to false	
bool([])	False
bool({})	False
'None' evaluates to false	
bool('none')	False

Logical operators

Operator	Example	True/False
and	True **and** True True **and** False False **and** True	True False False
'and' evaluates the first expression. If the first expression is false, the first expression is returned. Otherwise, the second expression is evaluated and is returned		
or	True or False False or True True or True False or False	True True True False
`**or**` only need one side to be `True`, so if the first expression is true that's what is returned. If the first expression is `False` then it moves to the second expression and returns that, no matter whether the second value evaluates to `True` or 'False'.		
not	**not** true **not** false	False True

Control flow and conditionals

if/elif/else

Operation	Example

if <condition> :
  <statement>
elif <condition> :
  <statement>
...
else:
  <statement>
	def greet_admin(user):
	  if user == "Guido":
	    return "Welcome, Guido."
	  elif user == "Bethany":
	    return "Welcome, Bethany."
	  elif user == "Alex":
	    return "Welcome, Alex."
	  else:
	    return "You are not authorized."

Exception handling

Operation

```
try:
    statements
  except [exception_type]: # (TypeError, ZeroDivisionError)
    statements
  else: # optional no exceptions
    statements
  finally: # optional all
    statements
```

Example

```
try:
   num1,num2=eval(input("Enter 2 numbers, using a comma"))
   result = num1 / num2
   print("Result is", result)
except ZeroDivisionError:
   print("Division by zero is error !!")
except SyntaxError:
   print("Comma is missing. Enter again with comma")
except:
   print("Wrong input")
else:
   print("No exceptions")
finally:
   print("This will execute no matter what")
```

Lists: Store a collection of data in an ordered sequence. List items can be of different types.

<table>
<tr><th>List activities</th><th>syntax</th></tr>
<tr><td>List creation</td><td>mylist = ['cats', 'dogs', 42, ['pizza', 'beer'], True]</td></tr>
<tr><td>Accessing list</td><td>mylist[0] returns cats</td></tr>
<tr><td>Update list</td><td>mylist[0] = 'bears' returns
['bears', 'dogs', 42, ['pizza', 'beer'], True]</td></tr>
<tr><td>Slicing list</td><td>
<table>
<tr><td>bears</td><td>dogs</td><td>42</td><td>['pizza', 'beer']</td><td>True</td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
</table>
mylist[1:4] returns ['dogs', 42, ['pizza', 'beer']]
mylist[2:] returns [42, ['pizza', 'beer'], True]
mylist[:2] returns ['bears', 'dogs']
mylist[2:-1] returns [42, ['pizza', 'beer']]
</td></tr>
</table>

List Methods

```
mylist = ['cats', 'dogs', 'birds’]
```

Method	syntax	returns
len(): Length of a list	`len(mylist)`	3
append(): Add an item to the end of the list	`mylist.append(‘pets’)`	['cats', 'dogs', 'birds', **'pets'**]
insert(): Add an item at a certain position in the list	`mylist.insert(1, ‘bears’)`	['cats', **'bears'**, 'dogs', 'birds, 'pets']
pop(): Removes and returns the last item on the list or the item at specified index	`mylist.pop()`	**pets** and the list changes to -> ['cats', 'bears', 'dogs', 'birds']
index(): To find the index of a matching item on the list	`mylist.index(‘dogs’)`	**2**
sort(): To sort a list	`mylist.sort()`	**['bears', 'birds', 'cats', 'dogs']**

Loops

While loop: statements execute as long as condition is true

while(expression):
stmts until expression is false
	`while n % 2 == 0:`
	`    print(n)`
	`    n = n // 2`

for loop: statements execute for each item in a sequence

for x in sequence:
#work on each member in the sequence. e.g., each item in a list, each character in a string
	`for character in "Howdy":`
	`    print(character)`
	
	H, o, w, d, y

for x in range(n):
#perform execution n times
	`for n in range(5):`
	`    print(n)`
	
	0,1,2,3,4

for x in range(a,b):
#perform execution starting at a and stopping at b
	`for num in range(10,15):`
	`    if (num % 2) == 0:`
	`      print('Even')`
	`    else:`
	`      print ('Odd')`
	
	Even, Odd, Even, Odd, Even

for x in range(a,b,c):
#perform execution starting at a and stopping at b, incrementing by c
	`for n in range(1,6,2):`
	`    print(n)`
	
	1,3,5

Dictionaries: Allows you to store data as an unordered collection of **key: value** pairs.

Dictionaries activities	syntax

Create dictionary
 	
dict = {key : value}
	`stock = {`
	`          "apples": 5,`
	`          "oranges": 2,`
	`          "pears": 10,`
	`        }`
	{'apples': 3, 'oranges': 2, 'pears': 10}

Modify dictionary
 	
dict[key1] = newValue
	`stock["apples"] -= 2`
	`stock["oranges"] = 20`
	`stock["kale"] = 20`
	
	{'apples': **3**, 'oranges': **20**, 'pears': 10, **'kale': 20**}

Delete element from dictionary	**del** stock["pears"] {'apples': 3, 'oranges': 20, 'kale': 20}

Dictionary Methods

stock = {"apples": 5, "oranges": 2, "pears": 10}

Method	syntax	returns
keys(): return all the keys in a dictionary	`stock.keys()`	dict_keys(['apples', 'oranges', 'pears'])
values(): return all the values in a dictionary	`stock.values()`	dict_values([5, 2, 10])
items(): return all the key:value pairs (or "items") in a dictionary	`stock.items()`	dict_items([('apples', 5), ('oranges', 2), ('pears', 10)])
clear(): remove all items from the dictionary	`stock.clear()`	{}

Objects, Classes, modules: Classes (and instances of classes, i.e. objects) encapsulate data and functions into self-contained bundles.

```
class Employee:
# __init__() is automatically called when an object
# is created
    def __init__(self, name, title, salary):
        self._name = name
        self._title  = title
        self._salary = salary

    def getName(self):
        return self._name

    def getTitle(self):
        return self._title

    def getSalary(self):
        return self._salary

    def setBonus(self, bonus):
        self.salary = self.salary + bonus

emp1 = Employee('Jane', 'CTO', 350000)
emp2 = Employee('John', 'Programmer', 85000)

print(emp1.getName())
emp1.setBonus(50000)
print(emp1.getSalary())

Jane
400000

print(emp2.getName())
emp2.setBonus(2000)
print(emp2.getSalary())

John
87000
```