Week 17 Study Guide

Table of Contents

- Python Learning Objectives (Day 1)
  - The None value
  - Boolean values
  - Truthiness
  - Number values
  - Integer
  - Float
  - Type casting
  - Arithmetic Operators
  - String values
  - Length
  - Indexing
  - String Functions
    - index
    - count
  - Concatenation
  - Formatting
  - Useful string methods
  - Variables
  - Duck typing
  - Assignment
  - Comparison operators
  - Assignment operators
  - Flow-control statements: if, while, for
  - `if-elif-else`
  - `for`
  - `while`, `break` and `continue`
  - Functions
  - Lambdas
  - Errors
- Python Learning Objectives (Day 2)
  - Functions
  - variable length positional arguments
  - variable length keyword arguments
  - Lists
  - Dictionaries
  - Sets
  - Tuples
  - Ranges
  - Built-in functions: filter, map, sorted, enumerate, zip, len, max, min, sum, any, all, dir
  - filter
  - map
  - sorted
  - enumerate
  - zip
  - len
  - max
  - min
  - sum
  - any
  - all
  - dir
  - Importing packages and modules
- Python Learning Objectives (Day 3)
  - Classes, methods, and properties
  - JavaScript to Python Classes cheat table
  - List comprehensions

Python Learning Objectives (Day 1)

The None value

This is the same as `null` in javaScript. It represents the lack of existence of a value. You must type it `None` with a capital letter.

Functions that do not return an explicit value, return `None` by default.

```
def print_hello(name):
    """
    This is a function which prints hello to a user, but does not
    return anything.
    """
    print(f"Hello, {name}")

value = print_hello('Bob') # value will be `None`
```

Boolean values

These work the same as in JavaScript, but you **must** capitalize `True` and `False`.

```
a = True
b = False

c = true # This will try to use a variable named `true`!
d = false  # This will try to use a variable named `false`!
```

The logical operators in Python read like English

Javascript	Python
&&	and
||	or
!	not

```
# Logical AND
print(True and True)    # => True
print(True and False)   # => False
print(False and False)  # => False

# Logical OR
print(True or True)     # => True
print(True or False)    # => True
print(False or False)   # => False

# Logical NOT
print(not True)             # => False
print(not False and True)   # => True
print(not True or False)    # => False
```

Truthiness

Everything is `True` unless it's one of these:

- `None`
- `False`
- `''`
- `[]`
- `()`
- `set()`
- `range(0)`

Number values

Integer

```
print(3)         # => 3
print(int(19))   # => 19
print(int())     # => 0
```

Float

```
print(2.24)      # => 2.24
print(2.)        # => 2.0
print(float())   # => 0.0
print(27e-5)     # => 0.00027
```

Type casting

You can convert (cast) numbers in python from one number type to another number type.

```
# Integer to Float
print(17)               # => 17
print(float(17))        # => 17.0

# Float to integer
print(17.0)             # => 17.0
print(int(17.0))        # => 17

# Float and integer to string
print(str(17.0) + ' and ' + str(17)) # => 17.0 and 17
```

Python does not automatically convert types like JavaScript does.

So this is an error

```
print(17.0 + ' and ' + 17)
# TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

Arithmetic Operators

Operator	JavaScript	Python
addition	`+`	`+`
subtraction	`-`	`-`
multiplication	`*`	`*`
division	`/`	`/`
modulo	`%`	`%`
exponent	`Math.pow()`	`**`
integer division		`//`

There is no `++` or `--` in Python.

String values

You can use both `'` and `"` for strings and escaping works the same as it does in JavaScript

```
# Escaping single quote
'Jodi asked, "What\'s up, Sam?"'
```

Triple quotes `'''` can be used for multiline strings.

```
print('''My instructions are very long so to make them
more readable in the code I am putting them on
more than one line. I can even include "quotes"
of any kind because they won't get confused with
the end of the string!''')
```

Both `'''` and `"""` work for these, but convention is to reserve `"""` for multiline comments and function docstrings.

```
def print_hello(name):
    """
    This is a docstring that explains what the function does
    It can be multiple lines, handy!
    You can use any combination of ' and " in these because
    python is looking for the ending triple " characters
    to determine the end.
    """
    print(f"Hello, {name}")
```

Length

```
print(len("Spaghetti")) # => 9
```

Indexing

```
# Normal indexing
print("Spaghetti"[0])    # => S
print("Spaghetti"[4])    # => h

# You can use negative indexes to start at the end.
print("Spaghetti"[-1])    # => i
print("Spaghetti"[-4])    # => e

# return a series of characters
print("Spaghetti"[1:4])  # => pag
print("Spaghetti"[4:-1])    # => hett
print("Spaghetti"[4:4])  # => (empty string)
print("Spaghetti"[:4])  # => Spag
print("Spaghetti"[:-1])    # => Spaghett
print("Spaghetti"[1:])  # => paghetti
print("Spaghetti"[-4:])    # => etti

# indexing past the beginning or end gives an error
print("Spaghetti"[15])    # => IndexError: string index out of range
print("Spaghetti"[-15])    # => IndexError: string index out of range

# but ranges past the beginning or end do not.
print("Spaghetti"[:15])    # => Spaghetti
print("Spaghetti"[15:])    # => (empty string)
print("Spaghetti"[-15:])    # => Spaghetti
print("Spaghetti"[:-15])    # => (empty string)
print("Spaghetti"[15:20])    # => (empty string)
```

String Functions

index

Similar to JavaScript's `indexOf` function

```
print("Spaghetti".index("h"))    # => 4
print("Spaghetti".index("t"))    # => 6
```

count

counts how many times a substring appears in a string

```
print("Spaghetti".count("h"))    # => 1
print("Spaghetti".count("t"))    # => 2
print("Spaghetti".count("s"))    # => 0
print('''We choose to go to the moon in this decade and do the other things,
not because they are easy, but because they are hard, because that goal will
serve to organize and measure the best of our energies and skills, because that
challenge is one that we are willing to accept, one we are unwilling to
postpone, and one which we intend to win, and the others, too.
'''.count('the '))               # => 4
```

Concatenation

You can use the `+` operator just like in JavaScript

```
print("gold" + "fish")    # => goldfish
```

You can use the `*` operator to repeat a string a given number of times

```
print("s"*5)             # => sssss
```

Formatting

```
first_name = "Billy"
last_name = "Bob"
# Using the format function
print('Your name is {0} {1}'.format(first_name, last_name))  # => Your name is Billy Bob
# Using the `f` format flag on the string
print(f'Your name is {first_name} {last_name}') # => Your name is Billy Bob
```

Useful string methods

Value	Method	Result
`s = "Hello"`	`s.upper()`	`"HELLO"`
`s = "Hello"`	`s.lower()`	`"hello"`
`s = "Hello"`	`s.islower()`	`False`
`s = "hello"`	`s.islower()`	`True`
`s = "Hello"`	`s.isupper()`	`False`
`s = "HELLO"`	`s.isupper()`	`True`
`s = "Hello"`	`s.startswith("He")`	`True`
`s = "Hello"`	`s.endswith("lo")`	`True`
`s = "Hello World"`	`s.split()`	`["Hello", "World"]`
`s = "i-am-a-dog"`	`s.split("-")`	`["i", "am", "a", "dog"]`

Method	Purpose
`isalpha()`	returns `True` if the string consists only of letters and is not blank.
`isalnum()`	returns `True` if the string consists only of letters and numbers and is not blank.
`isdecimal()`	returns `True` if the string consists only of numeric characters and is not blank.
`isspace()`	returns `True` if the string consists only of spaces, tabs, and newlines and is not blank.
`istitle()`	returns `True` if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.

Variables

Duck typing

If it looks like a duck and quacks like a duck, then it must be a duck.

Assignment

Just like JavaScript, but there are no special keywords. Scope is block scoped, much like `let` in JavaScript. You can also reassign variables just like `let`.

```
a = 7
b = 'Marbles'
print(a)         # => 7
print(b)         # => Marbles

# You can do assignment chaining
count = max = min = 0
print(count)           # => 0
print(max)             # => 0
print(min)             # => 0
```

Comparison operators

Python uses these same equality operators as JavaScript.

- `>` (greater than)
- `<` (less than)
- `>=` (greater than or equal to)
- `<=` (less than or equal to)
- `==` (equal to)
- `!=` (not equal to)

Precendence in Python:

- Negative signs (`not`) are applied first (part of each number)
- Multiplication and division (`and`) happen next
- Addition and subtraction (`or`) are the last step

Be careful using `not` along with `==`

```
print(not a == b)     # => True
# This breaks
print(a == not b)     # Syntax Error
# This fixes it
print (a == (not b))    # => False
```

Python does short-circuit evaluation

Expression	Right side evaluated?
`True` and …	Yes
`False` and …	No
`True` or …	No
`False` or …	Yes

Assignment operators

`=` is the normal assignment operator.

Python includes these assignment operators as well

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`
- `**=`
- `//=`

Flow-control statements: if, while, for

`if-elif-else`

```
if name == 'Monica':
    print('Hi, Monica.')
elif age < 12:
    print('You are not Monica, kiddo.')
else:
    print('You are neither Monica nor a little kid.')
```

`for`

```
# Looping over a string
for c in "abcdefg":
    print(c)

# Looping over a range
print('My name is')
for i in range(5):
   print('Carlita Cinco (' + str(i) + ')')

# Looping over a list
lst = [0, 1, 2, 3]
for i in lst:
    print(i)

# Looping over a dictionary
spam = {'color': 'red', 'age': 42}
for v in spam.values():
    print(v)

# Loop over a list of tuples and
# Destructuring to values
# Assuming spam.items returns a list of tuples
# Each containing two values (k, v)
for k, v in spam.items():
    print('Key: ' + k + ' Value: ' + str(v))
```

`while`, `break` and `continue`

`while` loop as long as the condition is `True`.

`break` allows you to break out of the loop.

`continue` skips this iteration of the loop and goes to the next iteration.

```
spam = 0
while True:
  print('Hello, world.')
  spam = spam + 1
  if spam < 5:
    continue
  break
```

Functions

You use the `def` keyword to define a function in Python.

```
# Basic function with no arguments and no return value
def printCopyright():
    print("Copyright 2020. Me, myself and I. All rights reserved.")

# Function with positional parameters and a return value
```

```python
def average(num1, num2):
    return (num1/num2)

# Calling it with positional arguments
print(average(6, 2))        # => 3.0

# Calling it with keyword arguments
# (note that order doesn't matter)
print(average(num2=2, num1=6));

# Default parameters
# Here the string "Hello" is the default for `saying`
def greeting(name, saying="Hello"):
    print(saying, name)

greeting("Monica") # => Hello Monica

greeting("Monica", saying="Hi") # => Hi Monica

# A common 'gotcha' is using an mutable object for a default parameter.
# Python doesn't do what you expect. All invocations of the function
# reference the same mutable object

# Everytime we call this we use the exact same `itemList` list
def appendItem(itemName, itemList = []):
    itemList.append(itemName)
    return itemList
print(appendItem('notebook')) # =>  ['notebook']
print(appendItem('pencil')) # =>  ['notebook', 'pencil']
print(appendItem('eraser')) # => ['notebook', 'pencil', 'eraser']
```

Lambdas

In python we have anonymous functions called lambdas, but they are only
a single python statement.

```python
toUpper = lambda s: s.upper()

toUpper('hello') # => HELLO
```

is the same as this in JavaScript

```javascript
const toUpper = s => s.toUpperCase();
toUpper('hello'); // # => HELLO
```

Errors

Unlike JavaScript, if you pass the wrong number of arguments to a function
it will throw an error.

```python
average(1)
# => TypeError: average() missing 1 required positional argument: 'num2'

average(1,2,3)
# => TypeError: average() takes 2 positional arguments but 3 were given
```

Python Learning Objectives (Day 2)

Functions

- `*` - Get the rest of the position arguments as a tuple
- `**` - Get the rest of the keyword arguments as a dictionary

variable length positional arguments

```python
def add(a, b, *args):
    # args is a tuple of the rest of the arguments
    total = a + b;
    for n in args:
        total += n
    return total

# args is None
add(1, 2)  # Returns 3

# args is (4, 5)
add(2, 3, 4, 5) # Returns 14
```

variable length keyword arguments

```python
def print_names_and_countries(greeting, **kwargs):
    # kwargs is a dictionary of the rest of the keyword arguments
    for k, v in kwargs.items():
        print(greeting, k, "from", v)

# kwargs would be:
# {
#   'Monica': 'Sweden',
#   'Charles'='British Virgin Islands',
#   'Carlo':'Portugal
# }
print_names_and_countries("Hi",
                          Monica="Sweden",
                          Charles="British Virgin Islands",
                          Carlo="Portugal")
# Prints
# Hi Monica from Sweden
# Hi Charles from British Virgin Islands
# Hi Carlo from Portugal
```

You can combine all of these together

```
def example2(arg_1, arg_2, *args, kw_1="shark", kw_2="blowfish", **kwargs):
  pass
```

Lists

Lists are mutable arrays.

```
# Can be made with square brackets
empty_list = []
departments = ['HR','Development','Sales','Finance','IT','Customer Support']

# list built-in function makes a list too
specials = list()

# You can use `in` to test if something is in the list
print(1 in [1, 2, 3]) #> True
print(4 in [1, 2, 3]) #> False
```

Dictionaries

Dictionaries are similar to JavaScript POJOs or `Map`. They have key value pairs.

```
# With curlies
a = {'one':1, 'two':2, 'three':3}
# With the dict built-in function
b = dict(one=1, two=2, three=3)

# You can use the `in` operator with dictionaries too
print(1 in {1: "one", 2: "two"})    #> True
print("1" in {1: "one", 2: "two"})  #> False
print(4 in {1: "one", 2: "two"})    #> False
```

Sets

Just like JavaScript's `Set`, it is an unordered collection of distinct objects.

```
# Using curlies (dont' confuse this with dictionaries)
school_bag = {'book','paper','pencil','pencil','book','book','book','eraser'}

# Using the set() built in
school_bag = set('book','paper','pencil','pencil','book','book','book','eraser')

# You can use the `in` operator with sets
print(1 in {1, 1, 2, 3})  #> True
print(4 in {1, 1, 2, 3})  #> False
```

Tuples

Tuples are *immutable* lists of items.

```
# With parenthesis
time_blocks = ('AM','PM')

# Without parenthesis
colors = 'red','blue','green'
numbers = 1, 2, 3

# with the tuple buit-in function which can also be used to
# convert things to tuples
tuple('abc')        # returns ('a', 'b', 'c')
tuple([1,2,3])      # returns (1, 2, 3)

# you can use the `in` operator with tuples
print(1 in (1, 2, 3)) #> True
print(4 in (1, 2, 3)) #> False
```

Ranges

A *range* is simply a list of numbers in order which can't be changed (immutable). Ranges are often used with `for` loops.

A `range` is declared using one to three parameters

- start - optional (`0` if not supplied) - first number in the sequence
- stop - required - next number past the last number in the sequence
- step - optional (`1` if not supplied) - the difference between each number in the sequence

For example

```
range(5)            # [0, 1, 2, 3, 4]
range(1,5)          # [1, 2, 3, 4]
range(0, 25, 5)     # [0, 5, 10, 15, 20]
range(0)            # [ ]
```

Built-in functions: filter, map, sorted, enumerate, zip, len, max, min, sum, any, all, dir

filter

```
def isOdd(num):
    return num % 2
filtered = filter(isOdd, [1,2,3,4])
# It returns a filter iterable object
# but we can cast it to a list
print(list(filtered)) # => [1, 3]
```

map

```
def toUpper(str):
    return str.upper()

upperCased = map(toUpper, ['a','b','c'])

print(list(upperCased)) # => ['A','B','C']
```

sorted

```
sortedItems = sorted(['Banana', 'orange', 'apple'])
print(list(sortedItems)) # => ['Banana', 'apple', 'orange']

# Notice Banana is first because uppercase letters come first

# Using a key function to control the sorting and make it sort
# so the case doesn't matter
sortedItems = sorted(['Banana', 'orange', 'apple'], key=str.lower)
print(list(sortedItems)) # => ['apple', 'Banana', 'orange']

# Reversing the sort
sortedItems = sorted(['Banana', 'orange', 'apple'], key=str.lower, reverse=True)
print(list(sortedItems)) # => ['orange', 'Banana', 'apple']
```

enumerate

```
quarters = ['First', 'Second', 'Third', 'Fourth']
print(enumerate(quarters))
print(enumerate(quarters, start=1))
```

```
(0, 'First'), (1, 'Second'), (2, 'Third'), (3, 'Fourth')
(1, 'First'), (2, 'Second'), (3, 'Third'), (4, 'Fourth')
```

zip

```
keys = ("Name", "Email")
values = ("Bob", "Bob@bob.com")

zipped = zip(keys, values)

print(list(zipped))
# => [('Name', 'Bob'), ('Email', 'Bob@bob.com')]

# You can zip more than two
x_coords = [0, 1, 2, 3, 4]
y_coords = [2, 3, 5, 3, 5]
z_coords = [3, 5, 2, 1, 4]

coords = zip(x_coords, y_coords, z_coords)

print(list(coords))
# => [(0, 2, 3), (1, 3, 5), (2, 5, 2), (3, 3, 1), (4, 5, 4)]
```

len

```
len([1,2,3]) # => 3
len((1,2,3)) # => 3
len({
    'Name': 'Bob',
    'Email': 'bob@bob.com'
}) # => 2
```

Can also work on any object which contains a `__len__` method.

max

```
max(1, 4, 6, 2) # => 6
max([1, 4, 6, 2]) # => 6
```

min

```
min(1, 4, 6, 2) # => 1
min([1, 4, 6, 2]) # => 1
```

sum

```
sum([1,2,3]) # => 6
```

any

```
any([True, False, False]) # => True
any([False, False, False]) # => False
```

all

```
any([True, False, False]) # => False
any([True, True, True]) # => False
```

dir

Returns all the attributes of an object including it's methods and dunder methods

```
user = {
    'Name': 'Bob',
    'Email': 'bob@bob.com'
}

dir(user)

# => ['__class__', '__contains__', '__delattr__', '__delitem__', '__dir__',
#  '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
# '__getitem__', '__gt__', '__hash__', '__init__', '__init_subclass__',
# '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__',
# '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__setattr__',
# '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'clear', 'copy',
# 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault',
# 'update', 'values']
```

Importing packages and modules

- Module - Python code in a file **or** directory
- Package - A module which is a directory containing a `__init__.py` file.
- submodule - A module which is contained within a package
- name - an exported function, class or variable in a module

Unlike JavaScript, modules export ALL names contained within them without any special `export` keywords

Assuming we have the following package with four submodules

```
math
|   __init.py
|   addition.py
|   subtraction.py
|   multiplication.py
|   division.py
```

if we peek into the addition.py file we see there's an add function

```
# addition.py
# We can import `add` from other places because it's a `name` and is
# AUTOMATICALLY exported
def add(num1, num2):
    return num1 + num2
```

Our `__init__.py` has the following lines:

```
# This imports the `add` function
# and now it's also re-exported in here as well!
from .addition import add
# These import and re-export the rest of the functions from the sub modules
from .subtraction import subtract
from .division import divide
from .multiplication import multiply
```

Remember any names that exist within a module are automatically exported.

> Notice the `.` syntax because this package can import it's own submodules.

So if we have a `script.py`, and we want to import the add function, we could do it lots of different ways

```
# This will load and execute the `math/__init__.py` file
# and give us an object with the exported names in `math/__init.py__`
import math

print(math.add(1,2)) # => 3
```

```
# This imports JUST the add from `math/__init__.py`
from math import add

print(add(1,2)) # => 3
```

```
# This skips importing from `math/__init__.py` (although it still runs)
# and imports directly from the addition.py file
from math.addition import add

print(add(1,2)) # => 3
```

```python
# this imports all the functions individually from `math/__init.py`
from math import add, subtract, multiply, divide

print(add(1,2)) # => 3
print(subtract(2, 1)) # => 1
```

```python
# This imports `add` and renames it to `addSomeNumbers`
from math import add as addSomeNumbers

print(addSomeNumbers(1, 2)) # => 3
```

Python Learning Objectives (Day 3)

Classes, methods, and properties

```python
class AngryBird:
    # Slots optimize property access and memory usage
    # and prevent you from arbitrarily assigning new properties to the instance
    __slots__ = ['_x', '_y']

    # constructor
    def __init__(self, x=0, y=0):
        # Doc string
        """
        Construct a new AngryBird by setting its position to (0, 0).
        """
        ## Instance variables
        self._x = x
        self._y = y

    # Instance method
    def move_up_by(self, delta):
        self._y += delta

    # Getter
    @property
    def x(self):
        return self._x

    # Setter
    @x.setter
    def x(self, value):
        if value < 0:
            value = 0
        self._x = value

    @property
    def y(self):
        return self._y

    @y.setter
    def y(self, new_y):
        self._y = new_y

    # Dunder Repr... called by `print`
    def __repr__(self):
        return f"<AngryBird ({self._x}, {self._y})>"
```

JavaScript to Python Classes cheat table

	Javascript	Python
Constructor	`constructor()`	`def __init__(self):`
Super Constructor	`super()`	`super().__init__()`
Instance properties	`this.property`	`self.property`
Calling Instance Methods	`this.method()`	`self.method()`
Defining Instance Methods	`method(arg1, arg2) {}`	`def method(self, arg1, arg2):`
Getter	`get someProperty() {}`	`@property`
Setter	`set someProperty() {}`	`@someProperty.setter`

List comprehensions

List comprehensions are a way to transform a list from one format to another.

They are a Pythonic alternative to using `map` or `filter`.

Syntax of a list comprehension:

`newList = [value loop condition]`

Using a for loop

```python
squares = []
for i in range(10):
    squares.append(i**2)

print(squares)
# Prints [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

You can change it to a list comprehension

```
# value = i ** 2
# loop = for i in range(10)
squares = [i**2 for i in range(10)]

print(list(squares))
# Prints [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

They can be used with a condition to do what `filter` does

```
sentence = 'the rocket came back from mars'
vowels = [character for character in sentence if character in 'aeiou']

print(vowels)
# Prints ['e', 'o', 'e', 'a', 'e', 'a', 'o', 'a']
```

You can also use them on dictionaries. We can use the `items()` method
for the dictionary to loop through it getting the keys and values out at once.

```
person = {
    'name': 'Corina',
    'age': 32,
    'height': 1.4
}

# This loops through and capitalizes the first letter of all the keys
newPerson = { key.title(): value for key, value in person.items() }
# Prints {'Name': 'Corina', 'Age': 32, 'Height': 1.4}
```