Python basics

These Jupyter Notebooks to present Python code. Execute blocks of code with **ctrl+enter**

Reading:

Python tutorial (https://docs.python.org/3.7/tutorial/) 3.1, 4.1 - 4.7, 5.1 - 5.6

Variable types

Python will infer the type when you define the variable. With `x = 2`, `x` is set to an **int**. With `x = 2.0`, `x` is set to a **float**.
`type(...)` displays a variable's type (useful for understanding the details of the code when debugging).

- **int**
  A (signed) integer ...−2,−1,0,1,2,...

```
In [ ]: x = 2
        print(type(x))
```

```
<class 'int'>
```

- **float**
  A real number with 16 digits of precision. (Equivalent to "double" in other languages. Python does not natively support single-precision floating point numbers.)

```
In [ ]: y = 2.0
        print(type(y))
```

```
<class 'float'>
```

- **str**
  A string, a sequence of 0 or more characters. Enclosed within a pair of single quotes `'` or a pair of double quotes `"`.

```
In [ ]: a = "Hello World!"
        print(type(a))
        print(a)
```

```
<class 'str'>
Hello World!
```

Exercise : Read 3.1.2 in the Python tutorial (https://docs.python.org/3.7/tutorial/introduction.html) and check the output of the following. (Double click the cell to see the actual code.)

- print("McDonald's")
- print("McDonald\'s")
- print('McDonald\'s')
- print('McDonald's')
- print(r"McDonald\'s")
- print(3*"McDonald\'s")
- print("McDonald" + "\'s")
- print("McDonald" "\'s")
- print("""McDonald

```
              \'s""")
```

```
In [ ]: print("""McDonald
            multiline string
                      \'s""")
```

```
McDonald
    multiline string
              's
```

Indices refer to positions **between** characters. The left edge of the first character numbered **0.** Then the right edge of the last character of a string of length **n** characters has index **n.**

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Python also uses negative indexes.

To access a single element of `seq` immediately after index `ind` , use `seq[ind]` .

Exercise: If `s = "abcdefg"` , what are the outputs to the following code?

- print(s[0])
- print(s[6])
- print(s[7])

```
In [ ]: s = "Python"
        print(s[:])
```

```
Python
```

The slice notation specifies two index positions separated by a colon (`:`) to access subsequences. `seq[start:stop]` returns elements in `seq` between `start` and `stop` . If `start` or `stop` are omitted, they are set to the beginning or the end. If `stop` is larger than the end of the string, `stop` is set to the end of the string.

Exercise: If `s = "abcdefg"` , what are the outputs to the following code?

- print(s[0])
- print(s[6])
- print(s[7])
- print(s[-1])
- print(s[1:3])
- print(s[:3])
- print(s[3:])
- print(s[:])
- print(s[0:-2])
- print(s[0:100])
- s[0] = 'z'
- s[0:3] = ['x','y','z']
- print(len(s))

In [ ]:
```
s[0] = 'z'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent ca
ll last)
/Users/movie/workspace/assignment-01-movie112/01_2.ipynb 셀 16 line
1
----> <a href='vscode-notebook-cell:/Users/movie/workspace/assignme
nt-01-movie112/01_2.ipynb#X21sZmlsZQ%3D%3D?line=0'>1</a> s[0] = 'z'

TypeError: 'str' object does not support item assignment
```

`seq[start:stop:step]` returns elements in `seq` between `start` and `stop` separated by `step` . (`step` is `1` by default.)

In [ ]:
```
s = 'Python'
print(s[::2])  # print elements in even-numbered positions
print(s[1::2])  # print elements in odd-numbered positions
print(s[::-1])  # print elements in reverse order
print(s[:-4:-1])
```

```
Pto
yhn
nohtyP
noh
```

Comments

Properly and adequately commenting is essential for ensuring your code is understandable by your colleagues and your future self. Make commenting a habit. (A wise man once said "We first make our habits, and then our habits make us.")
Python has two types of comments: long multi-line comments and short inline comments.

`#` is the comment character: anything after `#` on the line is ignored. It is considered good style to use at least 2 blank spaces before an inline comment following code.

Write **Multi-line comments** with multi-line strings, delimited by pairs of triple quotes (`'''` or `"""`).

In [ ]:
```
# this is single-line comment
print('Before comment')  #It's good style to use at least 2 spaces 
here before the #
'''
This is
a multi-line
comment
'''
print('After comment')
```

```
Before comment
After comment
```

Lists

A **list** is an ordered sequence of 0 or more comma-delimited elements enclosed within square brackets (`[` , `]`).

In [ ]:
```
s = [1,2,3,4,5,6,7,8]
print(s[2:5])
```

```
[3, 4, 5]
```

Use `+` to concatenate lists.

In [ ]:
```
s = ['a','b','c','d','e']+['f','g']
print(s)
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

Exercise: If `s = ['a','b','c','d','e'] + ['f','g']` , what are the outputs to the following code?

- print(s[0])
- print(s[6])
- print(s[7])
- print(s[-1])
- print(s[1:3])
- print(s[:3])
- print(s[3:])
- print(s[0:-2])
- print(s[0:100])
- s[0] = 'z'; print(s)
- s[0:3] = ['x','y','z']; print(s)

In [ ]:
```
s[0:3] = ['x','y','z']
print(s)
```

```
['x', 'y', 'z', 'd', 'e', 'f', 'g']
```

Lists may also have lists as their elements:

In [ ]:
```
K=[[1,2],[3,4,5],'s']
print(K[0][1])
print(K[0][:])
print(K[1])
print(K[2])
```

```
2
[1, 2]
[3, 4, 5]
s
```

The range function:

Generates sequences of numbers in the form of a list. The provided end point is not included. (Actually, `range` creates an iterable object, not a list. More on this later.)

In [ ]:
```
print(list(range(10)))
print(list(range(1,10)))
print(list(range(21,-1,-2)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[21, 19, 17, 15, 13, 11, 9, 7, 5, 3, 1]
```

If-else statement

You can write if-else statements with

```
if <condition>:
    code in if
elif <condition>:
    code in else-if
elif <condition>:
    code in else-if
else :
    code in else
code after if-else
```

The indentation for the if-else blocks is not optional.

In [ ]:
```
x = 0
if x > 0:
    print("x is positive")
elif x < 0:
    print("x is negative")
else:
    print("x is 0")

print("Code after if-else statement")
```

```
x is 0
Code after if-else statement
```

while loop

You can write while loop with

```
while <condition>:
    code in loop
code after loop
```

Again, indentation is not optional.

In [ ]:
```
var = 1
while var != "good bye" :
    var = input("Say something  :")
    print("You entered: {}".format(var))

print("Good bye!")
```

```
You entered:
You entered: good bye
Good bye!
```

for loop

You can write for loops with

```
for <var> in <list>:
    code in loop
    code in loop
code after loop
```

Again, indentation is not optional.

In [ ]:
```
# Print Fibonacci sequence
F_prev = 0
print(F_prev)
F_curr = 1
print(F_curr)

#for loop with numerical index
for i in range(10):
    F_next = F_prev + F_curr
    F_prev = F_curr
    F_curr = F_next
    print(F_next)
```

```
0
1
1
2
3
5
8
13
21
34
55
89
```

In [ ]:
```
#for loop with list
fruit_list = ["apple", "banana", "cherry"]
for fruit in fruit_list:
    print(fruit)
```

```
apple
banana
cherry
```

Generally, the for loop works with any "iterable" object. (More on this when we talk about objects.)

In [ ]:
```
#for loop with string
for c in "The Best Things in Life are Free":
    print("Current character: {}".format(c))
```

```
Current character: T
Current character: h
Current character: e
Current character:  
Current character: B
Current character: e
Current character: s
Current character: t
Current character:  
Current character: T
Current character: h
Current character: i
Current character: n
Current character: g
Current character: s
Current character:  
Current character: i
Current character: n
Current character:  
Current character: L
Current character: i
Current character: f
Current character: e
Current character:  
Current character: a
Current character: r
Current character: e
Current character:  
Current character: F
Current character: r
Current character: e
Current character: e
```

List comprehension

An intuitive and concise way to create lists.

In [ ]:
```
# ** power symbol
[i**2 for i in range(6)]
```

Out[ ]:
```
[0, 1, 4, 9, 16, 25]
```

In [ ]:
```
[0 for _ in range(5)]  #use _ if you don't need the variable
```

Out[ ]:
```
[0, 0, 0, 0, 0]
```

Use **if conditionals** to filter out elements

```
In [ ]: [i**2 for i in range(5) if i%2==0]
Out[ ]: [0, 4, 16]
```

You can use **multiple for clauses**

```
In [ ]: [i+j for i in range(2) for j in range(4)]
Out[ ]: [0, 1, 2, 3, 1, 2, 3, 4]
```

```
In [ ]: [i+j for j in range(4) for i in range(2)]  #What is the difference?
Out[ ]: [0, 1, 1, 2, 2, 3, 3, 4]
```

```
In [ ]: #For loop equivalent to the list comprehension
        L = []
        for i in range(2):
            for j in range(4):
                L.append(i+j)
        print(L)

        [0, 1, 2, 3, 1, 2, 3, 4]
```

```
In [ ]: [[i+j for i in range(2)] for j in range(4)]
Out[ ]: [[0, 1], [1, 2], [2, 3], [3, 4]]
```

```
In [ ]: [[i+j for j in range(4)] for i in range(2)]  #what there a differen
        ce?
Out[ ]: [[0, 1, 2, 3], [1, 2, 3, 4]]
```

Exercise : Use a list comprehension to create: [[1,2,3],[2,4,6],[3,6,9],[4,8,12]].

```
In [ ]: L = []
        for i in range(1, 5):
            l = []
            for j in range(1, 4):
                l.append(i*j)
            L.append(l)
        print(L)

        [[1, 2, 3], [2, 4, 6], [3, 6, 9], [4, 8, 12]]
```

Exercise : Use a list comprehension to create:[0,0,0,0,1,2,0,2,4,0,3,6,0,4,8,0,5,10].

```
In [ ]: L = []
        for i in range(6):
            for j in range(3):
                L.append(i*j)
        print(L)

        [0, 0, 0, 0, 1, 2, 0, 2, 4, 0, 3, 6, 0, 4, 8, 0, 5, 10]
```

Mutability

One important distinction between strings and lists is their **mutability** (https://docs.python.org/3.7/reference/datamodel.html).

Strings are **immutable**, i.e., they cannot be modified. Methods that seemingly do modify a given string (like `str.strip()`) return modified **copies**.

Lists are **mutable**, i.e., they can be modified.

The following examples show `list` methods modifying lists.

In [ ]:
```
list_1 = [1, 2, 3, 5, 1]
list_2 = list_1  # list_2 now references the same object as list_1

print('list_1:', list_1)
print('list_2:', list_2)

list_1.remove(1)  # remove [only] the first occurrence of 1 in list
_1
print('list_1:', list_1)
print('list_2:', list_2)

list_1.pop(2)  # remove the element in position 2
print('list_2:', list_2)

list_1.append(6)  # add 6 to the end of list_1
print('list_1:', list_1)

list_1.insert(0, 7)  # add 7 to the beinning of list_1 (before the
element in position 0)
print('list_1:', list_1)

list_1.sort()
print('list_1:', list_1)

list_1.reverse()
print('list_1:', list_1)

print('list_1:', list_1)
print('list_2:', list_2)
```

```
list_1: [1, 2, 3, 5, 1]
list_2: [1, 2, 3, 5, 1]
list_1: [2, 3, 5, 1]
list_2: [2, 3, 5, 1]
list_2: [2, 3, 1]
list_1: [2, 3, 1, 6]
list_1: [7, 2, 3, 1, 6]
list_1: [1, 2, 3, 6, 7]
list_1: [7, 6, 3, 2, 1]
list_1: [7, 6, 3, 2, 1]
list_2: [7, 6, 3, 2, 1]
```

The slice notation creates **a copy of a list**. In fact, using `[:]` is the standard way to create copies of the entire list.

If we assign that copy to another variable, the variables refer to different objects, so changes to one do not affect the other.

In [ ]:
```
list_1 = [1, 2, 3, 5, 1]
list_2 = list_1[:]  # list_1[:] returns a copy of the entire conten
ts of list_1

print('list_1:', list_1)
print('list_2:', list_2)

list_1.remove(1)  # remove [only] the first occurrence of 1 in list
_1
print('list_1:', list_1)
print('list_2:', list_2)
```

```
list_1: [1, 2, 3, 5, 1]
list_2: [1, 2, 3, 5, 1]
list_1: [2, 3, 5, 1]
list_2: [1, 2, 3, 5, 1]
```

Dictionary

A dictionary is a set of **keys** each pointing to a **value**. The keys must be unique, but values may be repeated.

In [ ]:
```
#d contains names and GPA key-value pairs. Duplicate names are not
allowed, but two people can have the same GPA.
d = {'Alice':4.23, 'Bob':4.1, 'Charlie': 3.8, 'Daniel':3.8}
print(d['Alice'])
print(d.keys())
print(d.values())
```

```
4.23
dict_keys(['Alice', 'Bob', 'Charlie', 'Daniel'])
dict_values([4.23, 4.1, 3.8, 3.8])
```

Modules

Modules are packages with classes and functions. You `import` a module to use it.

In [ ]:
```
import random  #import random module

#We can now use functions from the random module
print(random.random())  #uniform real in [0,1]
print(random.randint(1,3))  #uniform integer in {1,2,3}
L=[3,4,5]
random.shuffle(L)  #uniform shuffle of a list
print(L)
print(random.sample(L,2))  #uniform sampling
```

```
0.33357166348315603
2
[4, 5, 3]
[4, 3]
```

If you plan to use a module often, shorten the name with `as` .

In [ ]:
```
import random as rnd
rnd.random()
```

Out[ ]:
```
0.20478206327629755
```

You can import everything from a module and use the functions directly without naming the module. I think this is bad practice because we completely lose track of where the functions come from and, more importantly, the risk of name conflicts increase.

In [ ]:
```
from random import * #this is bad
from math import *

print(random())  #from which module does random come from?
log(5)  #Function logs (keeps record of) the number 5. (Just kiddin
g. It computes the natural logarithm.)
```

```
0.7361528636798355
```

Out[ ]:
```
1.6094379124341003
```

The previous code block did some bad things (for demonstration purposes). Let's clear everything with `%reset`

In [ ]:
```
%reset
```

Import specific features from a module with `from module import <thing>` and later use it without referring to the module name. This adds convenience without being reckless as `from module import *` .

In [ ]:
```
import numpy as np
from numpy.linalg import eigvals   #specifically import eigvals

A = np.matrix([[0, 0, 0, 0, 30/8],
               [1, 0, 0, 0, -67/8],
               [0, 1, 0, 0, -13/8],
               [0, 0, 1, 0, 54/8],
               [0, 0, 0, 1, 4/8]])


# print(np.linalg.eigvals(A))  # function call is a bit long and cu
mbersome
print(eigvals(A))  # we can call eigvals without referring to the m
odule name
```

```
[-2.  -1.5  0.5  1.   2.5]
```

Variables

In Python, **variables** are **names** that refer to **things** (such as objects).

"Changing something" can mean:

- changing what the name refers to (using `=`) or
- mutating the thing being referred to (using a member function).

In the following example, we use `id(obj)`, which returns a unique identifier of `obj`.

In [ ]:
```
#Variable is made to refer to something else
def f(x):
    print("x=",x," id=",id(x))
    x = 42
    print("x=",x," id=",id(x))

x = 5
print(x)
print(id(x))
f(x)
print(x)
print(id(x))
```

```
5
4339493344
x= 5  id= 4339493344
x= 42  id= 4339494528
5
4339493344
```

In [ ]:
```
#Variable refers to the same but mutated list
def f(lst):
    print("lst=",lst," id=",id(lst))
    lst.append(5)
    print("lst=",lst," id=",id(lst))

lst = ['a']
print(lst)
print(id(lst))
f(lst)
print(lst)
print(id(lst))

#Aside: don't use 'list' as a variable name since it is a reserved
Python keyword
```

```
['a']
140467719206848
lst= ['a']  id= 140467719206848
lst= ['a', 5]  id= 140467719206848
['a', 5]
140467719206848
```

Dintinguishing a variable from the object it refers to is essential.

In the code `x=[1,2,3]` :

- the right-hand-side `[1,2,3]` creates a list object and
- `x=` enters `x` into the "symbol table" (a table of recognized names) and makes `x` refer to the list object.

In the following example, `x == y` asks whether `x` and `y` represent the "same" thing while `x is y` asks whether `x` and `y` refer to the same object.

In [ ]:
```
x = [1,2,3]
y = [1,2,3]
z = x

print(x is y)
print(x == y)
print(x is z)
print(x == z)
print(y is z)
print(y == z)

x.append(4)
print(x)
print(y)
print(z)
```

```
False
True
True
True
False
True
[1, 2, 3, 4]
[1, 2, 3]
[1, 2, 3, 4]
```

Using `None` , a variable can be made to refer to nothing. A variable referring to `None` is not the same as a variable not exiting.

```
In [ ]: %reset
        x = None
        print(x is None)

        x = 6
        print(x is None)

        print(y is None)  #Error! y does not exist, so we cannot ask if it
        refers to anything.
```

```
True
False

---------------------------------------------------------------------------
NameError                                 Traceback (most recent ca
ll last)
/Users/movie/workspace/assignment-01-movie112/01_2.ipynb 셀 77 line
8
      <a href='vscode-notebook-cell:/Users/movie/workspace/assignme
nt-01-movie112/01_2.ipynb#Y136sZmlsZQ%3D%3D?line=4'>5</a> x = 6
      <a href='vscode-notebook-cell:/Users/movie/workspace/assignme
nt-01-movie112/01_2.ipynb#Y136sZmlsZQ%3D%3D?line=5'>6</a> print(x i
s None)
----> <a href='vscode-notebook-cell:/Users/movie/workspace/assignme
nt-01-movie112/01_2.ipynb#Y136sZmlsZQ%3D%3D?line=7'>8</a> print(y i
s None)  #Error! y does not exist, so we cannot ask if it refers to
anything.

NameError: name 'y' is not defined
```

Functions

A function is a block of code that runs when it is called. A function takes 0,1,2,... inputs and returns 0 or 1 result. A function can effectively return multiple objects by returning a list or a tuple.

```
In [ ]: #Define a function
        def nplusone(n):
            m = n + 1
            return m

        def isbig(n):
            if n > 10:
                return True
            else:
                return False

        #Call a function
        nplusone(6)

Out[ ]: 7
```

Exercise : Write a function `duplicates(lst)` that returns a list of all elements appearing twice or more in the input list `lst` .

```
In [ ]: def duplicates(lst):
            seen = set()
            duplicates = set()

            for item in lst:
                if item in seen:
                    duplicates.add(item)
                else:
                    seen.add(item)

            return list(duplicates)
```

Again, indentation is part of the formal syntax in Python and therefore is not optional.

```
In [ ]: #Error! Incorrect indentation!
        def nplusone(n):
        m = n + 1
        return m
```

```
  Cell In[67], line 3
    m = n + 1
    ^
IndentationError: expected an indented block
```

It is important to remember that mutable function inputs can be changed inside the function.

```
In [ ]: def f(x):
            x[1] = 1000
            return x

        def g(x):
            y = x[:] # creates a copy
            y[1] = 1000
            return y
```

```
In [ ]: a = [1, 2, 3]
        print("Initially, a was", a)
        f(a)
        print("Now, a is ",a)

        b = [1, 2, 3]
        print("Initially, b was", b)
        c = g(b)
        print("b is still",b)
        print("c is",c)
```

```
Initially, a was [1, 2, 3]
Now, a is  [1, 1000, 3]
Initially, b was [1, 2, 3]
b is still [1, 2, 3]
c is [1, 1000, 3]
```

In [ ]:
```
d = {'A':1, 'B':2}
print("Initially, d was", d)
f(d)
print("Now, d is", d)
```

```
Initially, d was {'A': 1, 'B': 2}
Now, d is {'A': 1, 'B': 2, 1: 1000}
```

Within functions, you have separate variable names.

In [ ]:
```
#Change within function not visible from outside
def f(x):
    x = 42  #x refers to something else, but the int object represe
nting 5 is still 5

x = 5
print(x)
f(x)
print(x)
```

```
5
5
```

In [ ]:
```
#Same program. The name of the function input y is separate from th
e
#name of x which was passed into the function.
def f(y):
    y = 42

x = 5
print(x)
f(x)
print(x)
```

```
5
5
```

In [ ]:
```
#Change within function visible from outside
def f(lst):
    lst.append(5)  #the list lst refers to is changed

lst = ['a']
print(lst)
f(lst)
print(lst)
```

```
['a']
['a', 5]
```

Functions with default argument values

Consider the function

```
In [ ]: def my_fun(a, b = 10, c = 20):
            print(a, b, c)
```

Predict the output of the following:

- my_fun( )
- my_fun(1)
- my_fun(1,2)
- my_fun(1,2,3)

```
In [ ]: my_fun(1,2,3)

        1 2 3
```

Important:

The default value for a function argument is only evaluated once, at the time that the function is defined.

Common mistake: misusing mutable default arguments

```
In [ ]: def foo(bar=[]):        # bar is optional and defaults to [] if not
        specified
            bar.append("SNU")    # but this line could be problematic, as w
        e'll see...
            return bar
```

```
In [ ]: foo([])

Out[ ]: ['SNU']
```

```
In [ ]: foo()

Out[ ]: ['SNU']
```

```
In [ ]: foo()

Out[ ]: ['SNU', 'SNU']
```

To fix this, we can do

```
In [ ]: def foo(bar = None):
            if bar is None:
                bar = []
            bar.append("SNU")
            return bar
```

```
In [ ]: foo()

Out[ ]: ['SNU']
```

In [ ]:
```
foo()
```
Out[ ]:
```
['SNU']
```

In [ ]:
```
foo()
```
Out[ ]:
```
['SNU']
```

Another common mistake: setting the current time as the default argument.

In [ ]:
```
from datetime import datetime

def printTime(currentTime = datetime.now()):
    print("The current time is " + str(currentTime))
```

In [ ]:
```
printTime()
```
```
The current time is 2024-03-14 22:32:24.256722
```

In [ ]:
```
printTime()
```
```
The current time is 2024-03-14 22:32:24.256722
```

Question: how would you fix `printTime` ?

Keyword Arguments:

Functions can also be called using **keyword arguments** of the form kwarg=value. (Normal arguments are called **positional** arguments.)

In [ ]:
```
def parrot(voltage, state = 'a stiff', action = 'voom', type = 'Nor
wegian Blue'):
    print("-- This parrot wouldn't", action)
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

In [ ]:
```
parrot(1000)                                          # 1 positiona
l argument
parrot(voltage = 1000)                                  # 1 keyword
argument
parrot(voltage = 1000000, action = 'VOOOOOM')             # 2 keywo
rd arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword a
rguments
parrot('a million', 'bereft of life', 'jump')         # 3 positiona
l arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positiona
l, 1 keyword
```

```
-- This parrot wouldn't voom
if you put 1000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't voom
if you put 1000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't VOOOOOM
if you put 1000000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't VOOOOOM
if you put 1000000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't jump
if you put a million volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's bereft of life !
-- This parrot wouldn't voom
if you put a thousand volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's pushing up the daisies !
```

In [ ]:
```
parrot(action='BAAM', 1000)   #ERROR! keyword arguments must come a
fter positional arguments
parrot(1000, foo=3)           #ERROR! Keyword arguments must match
one of the argumens
```

```
  Cell In[89], line 1
    parrot(action='BAAM', 1000)   #ERROR! keyword arguments must co
me after positional arguments
                              ^
SyntaxError: positional argument follows keyword argument
```

Exercise:

Read 4.7.3 and write a function **count_args** that accepts any number of input arguments and returns the number of arguments it received, e.g. count_args(10,2,3,1) returns 4 and count_args([10,2,3,1]) returns 1.

In [ ]:
```
def count_args(*args):
    return len(args)

count_args(10,2,3,1)
```

Out[ ]:
```
4
```

Unpacking argument list

You can unpack a list or tuple for a function with *.

In [ ]:
```
def fn(a, b, c):
    print(a + b + c)

fn(1, 2, 3)

lst = ["we", "love", "you"]
# fn(lst)  #fail
fn(*lst)

tpl = (1, 2, 7)
fn(*tpl)
```

```
6
weloveyou
10
```

You can use a dictionary to deliver keyword arguments for a function with **.

In [ ]:
```
def parrot(voltage, state, action):
    print("-- This parrot wouldn't", action)
    print("if you put", voltage, "volts through it.")
    print("E's", state, "!")

d = {"state": "bleedin' demised", "action": "VOOM", "voltage": "fou
r million"}
parrot(**d)
```

```
-- This parrot wouldn't VOOM
if you put four million volts through it.
E's bleedin' demised !
```

Tuples

Tuples are immutable, and usually contain a heterogeneous sequence of elements that are accessed via unpacking or indexing. **Lists** are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.

```
In [ ]: x = (3,'a',[1,2,3],{'A':1, 'B':2})
        a,b,c,d = x # tuple unpacking
        print(b)
        print(x[2])
        print(d)
```

```
a
[1, 2, 3]
{'A': 1, 'B': 2}
```

Multiple assignment

Using tuples, you can conveniently assign multiple variables in one line. This makes your code more concise and readable.

```
In [ ]: a,b = 1,2              #Multiple assignment without paratheses
        (c,d) = (4,"hello")    #Multiple assignment with paratheses
```

Although tuples are immutable, they can contain mutable objects

```
In [ ]: list1 = ['a','b']
        pair1 = (3,list1)
        list1.append('c')
        print(pair1)
        #pair1 did not change. It contains the same object references
        #the same list pair1 referes to changed
```

```
(3, ['a', 'b', 'c'])
```

Empty tuple and 1-tuple

You can have tuples of length 0 and 1

```
In [ ]: tpl = ()    #tuple of length 0
        print(tpl)

        tpl = (1,)  #tuple of length 1
        print(tpl)

        tpl = 1,    #tuple of length 1 (without parentheses)
        print(tpl)
```

```
()
(1,)
(1,)
```

Sets

A set is an unordered collection of items. Every element is unique (no duplicates) and must be immutable. However, the set itself is mutable. We can add or remove items from it.

In [ ]:
```
my_set = {1,2,3,4,3,2}

print(my_set)

set2 = set()
print(set2)

empty_dict = {}
print(empty_dict)
```

```
{1, 2, 3, 4}
set()
{}
```

In [ ]:
```
# my_set = {1, 2, [3,4]} # error! set cannot have mutable items
# my_set[0] # error! set does not support indexing
```

Try the following methods to change a set in Python:

- my_set = {1,2,3}
- my_set.add(4) # add one item
- my_set.update([5,6,7]) #add multiple items

In [ ]:
```
my_set = {1,2,3}
my_set.add(4)
my_set.update([5,6,7])
print(my_set)
```

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

Exercise: Determine the number of unique letters in "supercalifragilisticexpialidocious" using a set.

In [ ]:
```
s = "supercalifragilisticexpialidocious"
st = set()
for c in s:
    st.add(c)
print(len(st))
```

```
15
```

String formatting

String formatting is incredibly an incredibly convenient tool.

In [ ]:
```
person = {"name": "Alice", "age" : 20}

# Cumbersome to write
print('My name is '+ person["name"] + ' and I am ' + str(person["ag
e"]) + ' years old.')

# Much easier to write and read
print('My name is {} and I am {} years old.'.format(person["name"],
person["age"]))
```

```
My name is Alice and I am 20 years old.
My name is Alice and I am 20 years old.
```

There are many detailed features to string formatting. The following is just a few.

str.format

str.format() was introduced in Python 2.6

For more information, see https://docs.python.org/3.7/library/string.html#formatstrings (https://docs.python.org/3.7/library/string.html#formatstrings)

In [ ]:
```
person = {"name": "Alice", "age" : 20}

#Use numbers to refer to positional arguments
print('My name is {0} and I am {1} years old.'.format(person["nam
e"],person["age"]))
print('I am {1} years old and my name is {0}.'.format(person["nam
e"],person["age"]))

#You can use keyword arguments
print('My phone number is : {areaCode}{sep}{centralCode}{sep}{stati
onCode}'.format(areaCode=310, centralCode=111, stationCode=312, sep
="-"))

#You can access list and dictionary entries with [..]
print('My name is {0[name]} and I am {0[age]} years old.'.format(pe
rson))

import math

radius = 2.2
print('The circle with radius {} has area {:.2f}'.format(radius, ra
dius**2*math.pi))
```

```
My name is Alice and I am 20 years old.
I am 20 years old and my name is Alice.
My phone number is : 310-111-312
My name is Alice and I am 20 years old.
The circle with radius 2.2 has area 15.21
```

f-strings

f-string is a string manipulation tool introduced in Python 3.6.

In [ ]:
```
name = "Eric"
age = 74
print(f"Hello, {name}. You are {age} years old.")



weight = 150
print(f"{name} weighs {weight} pounds.")
print(F"{name} weighs {weight*0.45359237} kilograms.")
print(f"{name} weighs {weight*0.45359237:.2f} kilograms.")
```

```
Hello, Eric. You are 74 years old.
Eric weighs 150 pounds.
Eric weighs 68.0388555 kilograms.
Eric weighs 68.04 kilograms.
```

Useful built-in functions for manipulating sequence elements

```
- enumerate
- zip
- reversed
- items
```

`enumerate` adds a counter to an iterable.

In [ ]:
```
l = ['tic', 'tac', 'toe']

for i,v in enumerate(l,1):
    print(i,v)
```

```
1 tic
2 tac
3 toe
```

`zip` combines lists (and iterables) into an iterable of tuples.

In [ ]:
```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']


for (q, a) in zip(questions, answers):
    print('What is your {0}?  It is {1}.'.format(q, a))

for q, a in zip(questions, answers):  # explicit paratheses (..) ca
n be omitted
    print('What is your {0}?  It is {1}.'.format(q, a))
```

```
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

`reversed` reverses an iterable.

In [ ]:
```
l = list(range(1,10,2))
for i in reversed(l):
    print (i)

# you can achieve the same effect with slice indexing
for i in l[::-1]:
    print (i)
```

```
9
7
5
3
1
9
7
5
3
1
```

`items` returns an iterable containing key value pairs of a dictionary.

In [ ]:
```
month_name = {1: 'Jan', 2: 'Feb', 3:'Mar'}
for k, v in month_name.items():
    print(k, v)
```

```
1 Jan
2 Feb
3 Mar
```

The del statement

`del` removed variables and elements of lists

In [ ]:
```
s = "hello"
del s
# print(s)  # reference variable deletes

a = list(range(5))
print(a)

del a[2]
print(a)

del a[1:3]
print(a)

del a[:]
print(a)

del a
# print(a)
```

```
[0, 1, 2, 3, 4]
[0, 1, 3, 4]
[0, 4]
[]
```

More precisely, `del` deletes the reference variable, not necessarily the object itself.

In [ ]:
```
A = "hello"
B = A
del A
print(B)  #No error. A is deleted but the object "hello" still exis
ts as it is still referenced by B
```

```
hello
```