The Python Programming Language: Functions

In [4]:
```
x = 1
y = 2
print(x + y)
x - y
```

```
3
-1
```

Out[4]:

In [2]:
```
y
```

Out[2]:
```
2
```

`add_numbers` is a function that takes two numbers and adds them together.

In [6]:
```
def add_numbers(x, y):
    return x + y


add_numbers(1, 2)
```

Out[6]:
```
3
```

`add_numbers` updated to take an optional 3rd parameter. Using `print` allows printing of multiple expressions within a single cell.

In [8]:
```
def add_numbers(x=1, y=2, z=None):
    if (z == None):
        return x + y
    else:
        return x + y + z


print(add_numbers(1, 2))
print(add_numbers(1, 2, 3))
```

```
3
6
```

`add_numbers` updated to take an optional flag parameter.

In [ ]:
```
def add_numbers(x, y, z=None, flag=False):
    if (flag):
        print('Flag is true!')
    if (z == None):
        return x + y
    else:
        return x + y + z


print(add_numbers(1, 2, flag=True))
```

Assign function `add_numbers` to variable `a`.

In [9]:
```
def add_numbersx(, y):
    return x + y


a = add_numbers
a(1, 2)
```

Out[9]:
```
3
```

The Python Programming Language: Types and Sequences

Use `type` to return the object's type.

In [10]:
```
type('This is a string')
```

Out[10]:
```
str
```

In [11]:
```
type(None)
```

Out[11]:
```
NoneType
```

In [12]:
```
type(1)
```

Out[12]:
```
int
```

In [13]:
```
type(1.0)
```

Out[13]:
```
float
```

In [14]:
```
type(add_numbers)
```

Out[14]:
```
function
```

Tuples are an immutable data structure (cannot be altered).

In [20]:
```
x = ('a', 1, 2, 'b')
type(x)
x['a']
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_29112\1247444608.py in <module>
      1 x = ('a', 1, 2, 'b')
      2 type(x)
----> 3 x['a']

TypeError: tuple indices must be integers or slices, not str
```

Lists are a mutable data structure.

In [22]:
```
x = ['a', 1, 2, 'b']
type(x)
x[0]
```

Out[22]:
```
'a'
```

Use `append` to append an object to a list.

In [25]:
```
x.append(3.3)
print(x)
```

```
['a', 1, 2, 'b', 3.3, 3.3, 3.3]
```

This is an example of how to loop through each item in the list.

In [26]:
```
for item in x:
    print(item)
```

```
a
1
2
b
3.3
3.3
3.3
```

Or using the indexing operator:

In [27]:
```
i = 0
while (i != len(x)):
    print(x[i])
    i = i + 1
```

```
a
1
2
b
3.3
3.3
3.3
```

Use `+` to concatenate lists.

In [28]:
```
[1, 2] + [3, 4]
```

Out[28]:
```
[1, 2, 3, 4]
```

Use `*` to repeat lists.

In [29]:
```
[1] * 3
```

Out[29]:
```
[1, 1, 1]
```

Use the `in` operator to check if something is inside a list.

In [30]:
```
1 in [1, 2, 3]
```

Out[30]:
```
True
```

Now let's look at strings. Use bracket notation to slice a string.

In [34]:
```
x = 'This is a string'
print(x[0])  #first character
```

```
print(x[0:1])  #first character, but we have explicitly set the end character
print(x[0:5])  #first two characters
```

```
T
T
This
```

This will return the last element of the string.

In [36]:
```
x[-2]
```

Out[36]:
```
'n'
```

This will return the slice starting from the 4th element from the end and stopping before the 2nd element from the end.

In [37]:
```
x[-4:-2]
```

Out[37]:
```
'ri'
```

This is a slice from the beginning of the string and stopping before the 3rd element.

In [39]:
```
x[:3]
```

Out[39]:
```
'Thi'
```

And this is a slice starting from the 4th element of the string and going all the way to the end.

In [40]:
```
x[3:]
```

Out[40]:
```
's is a string'
```

In [43]:
```
firstname = 'Christopher'
lastname = 'Brooks'

print(firstname + ' ' + lastname)
print(firstname * 3)
print('Chris' in firstname)
```

```
Christopher Brooks
ChristopherChristopherChristopher
True
```

`split` returns a list of all the words in a string, or a list split on a specific character.

In [46]:
```
firstname = 'Christopher Arthur Hansen Brooks'.split(' ')[0]  # [0] selects the fi
lastname = 'Christopher Arthur Hansen Brooks'.split(' ')[-1]  # [-1] selects the l
print(firstname)
print(lastname)
```

```
Christopher
Brooks
```

Make sure you convert objects to strings before concatenating.

In [47]:
```
'Chris' + 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_29112\3401447053.py in <module>
----> 1 'Chris' + 2

TypeError: can only concatenate str (not "int") to str
```

In [48]:
```
'Chris' + str(2)
```

Out[48]:
```
'Chris2'
```

Dictionaries associate keys with values.

In [49]:
```
x = {'Christopher Brooks': 'brooksch@umich.edu', 'Bill Gates': 'billg@microsoft.co
x['Christopher Brooks']  # Retrieve a value by using the indexing operator
```

Out[49]:
```
'brooksch@umich.edu'
```

In [54]:
```
x['Christopher Brooks'] = 'brooksch@mich.edu'
x
```

Out[54]:
```
{'Christopher Brooks': 'brooksch@mich.edu',
 'Bill Gates': 'billg@microsoft.com',
 'Kevyn Collins-Thompson': 'kct@sm.com',
 'Kevyn Thompson': 'kct@sm.com'}
```

Iterate over all of the keys:

In [58]:
```
for i, name in enumerate(x):
    print(i,name, x[name])
```

```
0 Christopher Brooks brooksch@mich.edu
1 Bill Gates billg@microsoft.com
2 Kevyn Collins-Thompson kct@sm.com
3 Kevyn Thompson kct@sm.com
```

Iterate over all of the values:

In [59]:
```
for email in x.values():
    print(email)
```

```
brooksch@mich.edu
billg@microsoft.com
kct@sm.com
kct@sm.com
```

Iterate over all of the items in the list:

In [60]:
```
for name, email in x.items():
    print(name)
    print(email)
```

```
Christopher Brooks
brooksch@mich.edu
Bill Gates
billg@microsoft.com
Kevyn Collins-Thompson
kct@sm.com
Kevyn Thompson
kct@sm.com
```

You can unpack a sequence into different variables:

In [63]:
```
x = ('Christopher', 'Brooks', 'brooksch@umich.edu')
fname, lname, email = x
```

In [64]:
```
fname
```

Out[64]:
```
'Christopher'
```

In [65]:
```
lname
```

Out[65]:
```
'Brooks'
```

Make sure the number of values you are unpacking matches the number of variables being assigned.

In [67]:
```
x = ('Christopher', 'Brooks', 'brooksch@umich.edu', 'Ann Arbor')
fname, lname, email, name = x
```

The Python Programming Language: More on Strings

In [68]:
```
print('Chris' + 2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_29112\960420689.py in <module>
----> 1 print('Chris' + 2)

TypeError: can only concatenate str (not "int") to str
```

In [69]:
```
print('Chris' + str(2))
```

```
Chris2
```

Python has a built in method for convenient string formatting.

In [ ]:
```
sales_record = {
    'price': 3.24,
    'num_items': 4,
    'person': 'Chris'}

sales_statement = '{} bought {} item(s) at a price of {} each for a total of {}'

print(sales_statement.format(sales_record['person'],
                             sales_record['num_items'],
                             sales_record['price'],
                             sales_record['num_items'] * sales_record['price']))
```

Reading and Writing CSV files

Let's import our datafile mpg.csv, which contains fuel economy data for 234 cars.

- mpg : miles per gallon

- class : car classification
- cty : city mpg
- cyl : # of cylinders
- displ : engine displacement in liters
- drv : f = front-wheel drive, r = rear wheel drive, 4 = 4wd
- fl : fuel (e = ethanol E85, d = diesel, r = regular, p = premium, c = CNG)
- hwy : highway mpg
- manufacturer : automobile manufacturer
- model : model of car
- trans : type of transmission
- year : model year

In [ ]:
```
import csv

%precision 2

with open('datasets/mpg.csv') as csvfile:
    mpg = list(csv.DictReader(csvfile))

mpg[:3]  # The first three dictionaries in our list.
```

`csv.Dictreader` has read in each row of our csv file as a dictionary. `len` shows that our list is comprised of 234 dictionaries.

In [ ]:
```
len(mpg)
```

`keys` gives us the column names of our csv.

In [ ]:
```
mpg[0].keys()
```

This is how to find the average cty fuel economy across all cars. All values in the dictionaries are strings, so we need to convert to float.

In [ ]:
```
sum(float(d['cty']) for d in mpg) / len(mpg)
```

Similarly this is how to find the average hwy fuel economy across all cars.

In [ ]:
```
sum(float(d['hwy']) for d in mpg) / len(mpg)
```

Use `set` to return the unique values for the number of cylinders the cars in our dataset have.

In [ ]:
```
cylinders = set(d['cyl'] for d in mpg)
cylinders
```

Here's a more complex example where we are grouping the cars by number of cylinder, and finding the average cty mpg for each group.

In [ ]:
```
CtyMpgByCyl = []

for c in cylinders:  # iterate over all the cylinder levels
    summpg = 0
    cyltypecount = 0
```

```
    for d in mpg:  # iterate over all dictionaries
        if d['cyl'] == c:  # if the cylinder level type matches,
            summpg += float(d['cty'])  # add the cty mpg
            cyltypecount += 1  # increment the count
    CtyMpgByCyl.append((c, summpg / cyltypecount))  # append the tuple ('cylinder'

CtyMpgByCyl.sort(key=lambda x: x[0])
CtyMpgByCyl
```

Use `set` to return the unique values for the class types in our dataset.

In [ ]:
```
vehicleclass = set(d['class'] for d in mpg)  # what are the class types
vehicleclass
```

And here's an example of how to find the average hwy mpg for each class of vehicle in our dataset.

In [ ]:
```
HwyMpgByClass = []

for t in vehicleclass:  # iterate over all the vehicle classes
    summpg = 0
    vclasscount = 0
    for d in mpg:  # iterate over all dictionaries
        if d['class'] == t:  # if the cylinder amount type matches,
            summpg += float(d['hwy'])  # add the hwy mpg
            vclasscount += 1  # increment the count
    HwyMpgByClass.append((t, summpg / vclasscount))  # append the tuple ('class',

HwyMpgByClass.sort(key=lambda x: x[1])
HwyMpgByClass
```

The Python Programming Language: Dates and Times

In [ ]:
```
import datetime as dt
import time as tm
```

`time` returns the current time in seconds since the Epoch. (January 1st, 1970)

In [ ]:
```
tm.time()
```

Convert the timestamp to datetime.

In [ ]:
```
dtnow = dt.datetime.fromtimestamp(tm.time())
dtnow
```

Handy datetime attributes:

In [ ]:
```
dtnow.year, dtnow.month, dtnow.day, dtnow.hour, dtnow.minute, dtnow.second  # get
```

`timedelta` is a duration expressing the difference between two dates.

In [ ]:
```
delta = dt.timedelta(days=100)  # create a timedelta of 100 days
delta
```

`date.today` returns the current local date.

In [ ]:
```
today = dt.date.today()
```

In [ ]:
```
today - delta  # the date 100 days ago
```

In [ ]:
```
today > today - delta  # compare dates
```

The Python Programming Language: Objects and map()

An example of a class in python:

In [ ]:
```
class Person:
    department = 'School of Information'  #a class variable

    def set_name(self, new_name):  #a method
        self.name = new_name

    def set_location(self, new_location):
        self.location = new_location
```

In [ ]:
```
person = Person()
person.set_name('Christopher Brooks')
person.set_location('Ann Arbor, MI, USA')
print('{} live in {} and works in the department {}'.format(person.name, person.lo
```

Here's an example of mapping the `min` function between two lists.

In [ ]:
```
store1 = [10.00, 11.00, 12.34, 2.34]
store2 = [9.00, 11.10, 12.34, 2.01]
cheapest = map(min, store1, store2)
cheapest
```

Now let's iterate through the map object to see the values.

In [ ]:
```
for item in cheapest:
    print(item)
```

The Python Programming Language: Lambda and List Comprehensions

Here's an example of lambda that takes in three parameters and adds the first two.

In [ ]:
```
my_function = lambda a, b, c: a + b
```

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

Let's iterate from 0 to 999 and return the even numbers.

In [ ]:
```
my_list = []
for number in range(0, 1000):
    if number % 2 == 0:
        my_list.append(number)
my_list
```

Now the same thing but with list comprehension.

In [ ]:
```
my_list = [number for number in range(0, 1000) if number % 2 == 0]
my_list
```