arm - 2024-02February-14

February 15, 2024

1 Agenda

1. Data structures
2. Functions
   - Arguments and parameters
   - Bytecodes
   - Scoping
   - Inner functions
   - Type hints/annotations + Mypy
3. Objects
   - Class, instance, method
   - Attributes – ICPO
   - Inheritance
   - Magic methods
   - Properties + descriptors
   - Memory management + objects
4. Iterators and generators
5. Concurrency – thread, processes, asyncio
6. pytest

2 Assignment

[2]:
```
x = 5
print(x)
```

```
5
```

[3]:
```
type(x)
```

[3]:
```
int
```

[4]:
```
x = 'abcd'
type(x)
```

[4]:
```
str
```

[5]:
```
x = 10
y = x
```

```
x = 20
print(y)
```

```
10
```

[6]:
```
x = 10
id(x)
```

[6]:
```
4387289840
```

[7]:
```
x += 1
id(x)
```

[7]:
```
4387289872
```

[ ]:
```
x = [10, 20, 30]
y = x

x[0] = 999
print(y)

x.append(40)
print(y)
```

[8]:
```
# I can ask if two values are equal

x = 10
y = 10

x == y
```

[8]:
```
True
```

[9]:
```
# are they  the same object?

x is y     # id(x) == id(y)
```

[9]:
```
True
```

[10]:
```
x = 1000
y = 1000

x == y
```

[10]:
```
True
```

[11]:
```
x is y
```

[11]:
```
False
```

[12]:
```
x = None

if x == None:    # not Pythonic!
    print('Yes, it is None!')
```

```
Yes, it is None!
```

[13]:
```
if x is None:
    print('Yes, it is None')
```

```
Yes, it is None
```

[14]:
```
x = 'abcd'
y = 'abcd'

x == y
```

[14]:
```
True
```

[15]:
```
x is y
```

[15]:
```
True
```

[16]:
```
x = 'ab.cd'
y = 'ab.cd'

x == y
```

[16]:
```
True
```

[17]:
```
x is y
```

[17]:
```
False
```

[18]:
```
i = 10
```

[19]:
```
globals()
```

[19]:
```
{'__name__': '__main__',
 '__doc__': 'Automatically created module for IPython interactive environment',
 '__package__': None,
 '__loader__': None,
 '__spec__': None,
 '__builtin__': <module 'builtins' (built-in)>,
 '__builtins__': <module 'builtins' (built-in)>,
 '_ih': ['',
  '2+3',
```

```
 'x = 5\nprint(x)',
 'type(x)',
 "x = 'abcd'\ntype(x)",
 'x = 10\ny = x   \n\nx = 20\nprint(y)',
 'x = 10\nid(x)  ',
 'x += 1  \nid(x)',
 '# I can ask if two values are equal\n\nx = 10\ny = 10\n\nx == y ',
 '# are they  the same object?\n\nx is y     # id(x) == id(y)',
 'x = 1000\ny = 1000\n\nx == y',
 'x is y',
 "x = None\n\nif x == None:\n    print('Yes, it is None!')",
 "if x is None:   \n    print('Yes, it is None')",
 "x = 'abcd'\ny = 'abcd'\n\nx == y",
 'x is y',
 "x = 'ab.cd'\ny = 'ab.cd'\n\nx == y",
 'x is y',
 'i = 10',
 'globals()'],
'_oh': {1: 5,
 3: int,
 4: str,
 6: 4387289840,
 7: 4387289872,
 8: True,
 9: True,
 10: True,
 11: False,
 14: True,
 15: True,
 16: True,
 17: False},
'_dh': [PosixPath('/Users/reuven/Courses/Current/arm-2024-q1-arm')],
'In': ['',
 '2+3',
 'x = 5\nprint(x)',
 'type(x)',
 "x = 'abcd'\ntype(x)",
 'x = 10\ny = x   \n\nx = 20\nprint(y)',
 'x = 10\nid(x)  ',
 'x += 1  \nid(x)',
 '# I can ask if two values are equal\n\nx = 10\ny = 10\n\nx == y ',
 '# are they  the same object?\n\nx is y     # id(x) == id(y)',
 'x = 1000\ny = 1000\n\nx == y',
 'x is y',
 "x = None\n\nif x == None:\n    print('Yes, it is None!')",
 "if x is None:   \n    print('Yes, it is None')",
 "x = 'abcd'\ny = 'abcd'\n\nx == y",
```

```
  'x is y',
  "x = 'ab.cd'\ny = 'ab.cd'\n\nx == y",
  'x is y',
  'i = 10',
  'globals()'],
 'Out': {1: 5,
  3: int,
  4: str,
  6: 4387289840,
  7: 4387289872,
  8: True,
  9: True,
  10: True,
  11: False,
  14: True,
  15: True,
  16: True,
  17: False},
 'get_ipython': <bound method InteractiveShell.get_ipython of
<ipykernel.zmqshell.ZMQInteractiveShell object at 0x107ae25a0>>,
 'exit': <IPython.core.autocall.ZMQExitAutocall at 0x107bd3590>,
 'quit': <IPython.core.autocall.ZMQExitAutocall at 0x107bd3590>,
 'open': <function _io.open(file, mode='r', buffering=-1, encoding=None,
errors=None, newline=None, closefd=True, opener=None)>,
 '_': False,
 '__': True,
 '___': True,
 '__session__': '/Users/reuven/Courses/Current/arm-2024-q1-arm/arm -
2024-02February-14.ipynb',
 '_i': 'i = 10',
 '_ii': 'x is y',
 '_iii': "x = 'ab.cd'\ny = 'ab.cd'\n\nx == y",
 '_i1': '2+3',
 '_1': 5,
 '_i2': 'x = 5\nprint(x)',
 'x': 'ab.cd',
 '_i3': 'type(x)',
 '_3': int,
 '_i4': "x = 'abcd'\ntype(x)",
 '_4': str,
 '_i5': 'x = 10\ny = x   \n\nx = 20\nprint(y)',
 'y': 'ab.cd',
 '_i6': 'x = 10\nid(x)  ',
 '_6': 4387289840,
 '_i7': 'x += 1  \nid(x)',
 '_7': 4387289872,
 '_i8': '# I can ask if two values are equal\n\nx = 10\ny = 10\n\nx == y ',
```

```
'_8': True,
'_i9': '# are they  the same object?\n\nx is y     # id(x) == id(y)',
'_9': True,
'_i10': 'x = 1000\ny = 1000\n\nx == y',
'_10': True,
'_i11': 'x is y',
'_11': False,
'_i12': "x = None\n\nif x == None:\n    print('Yes, it is None!')",
'_i13': "if x is None:   \n    print('Yes, it is None')",
'_i14': "x = 'abcd'\ny = 'abcd'\n\nx == y",
'_14': True,
'_i15': 'x is y',
'_15': True,
'_i16': "x = 'ab.cd'\ny = 'ab.cd'\n\nx == y",
'_16': True,
'_i17': 'x is y',
'_17': False,
'_i18': 'i = 10',
'i': 10,
'_i19': 'globals()'}
```

[20]:
```
x = 'abcd'
id(x)
```

[20]: 4425116688

[23]:
```
# interning

import sys

id(sys.intern('abcd'))
```

[23]: 4425116688

[24]:
```
id(sys.intern('abcd'))
```

[24]: 4425116688

[25]:
```
id(sys.intern('ab.cd'))
```

[25]: 4425512256

[26]:
```
id(sys.intern('ab.cd'))
```

[26]: 4425512256

[27]:
```
x = 'ab.cd'
id(x)
```

[27]: 4425509040

[28]:
```
x = sys.intern('ab.cd')
```

[29]:
```
id(x)
```

[29]: 4425512256

[30]:
```
import sys
sys.version
```

[30]: '3.12.1 (main, Jan 29 2024, 07:04:51) [Clang 15.0.0 (clang-1500.1.0.2.5)]'

[31]:
```
i = 0

sys.getsizeof(i)
```

[31]: 28

[32]:
```
i = 1
sys.getsizeof(i)
```

[32]: 28

[33]:
```
i = 1234567890
sys.getsizeof(i)
```

[33]: 32

[34]:
```
i = i ** 1000
```

[35]:
```
sys.getsizeof(i)
```

[35]: 4052

[36]:
```
mylist = []
sys.getsizeof(mylist)
```

[36]: 56

[37]:
```
mylist = [10, 20, 30, 40, 50]
sys.getsizeof(mylist)
```

[37]: 104

[38]:
```
mylist[0] = 'abcdefghij' * 100_000
```

[39]:
```
sys.getsizeof(mylist)
```

[39]: 104

[40]:
```
mylist = []

for i in range(30):
    print(f'{i=}, {len(mylist)=}, {sys.getsizeof(mylist)=}')    # format string / fancy string
    mylist.append(i)
```

```
i=0, len(mylist)=0, sys.getsizeof(mylist)=56
i=1, len(mylist)=1, sys.getsizeof(mylist)=88
i=2, len(mylist)=2, sys.getsizeof(mylist)=88
i=3, len(mylist)=3, sys.getsizeof(mylist)=88
i=4, len(mylist)=4, sys.getsizeof(mylist)=88
i=5, len(mylist)=5, sys.getsizeof(mylist)=120
i=6, len(mylist)=6, sys.getsizeof(mylist)=120
i=7, len(mylist)=7, sys.getsizeof(mylist)=120
i=8, len(mylist)=8, sys.getsizeof(mylist)=120
i=9, len(mylist)=9, sys.getsizeof(mylist)=184
i=10, len(mylist)=10, sys.getsizeof(mylist)=184
i=11, len(mylist)=11, sys.getsizeof(mylist)=184
i=12, len(mylist)=12, sys.getsizeof(mylist)=184
i=13, len(mylist)=13, sys.getsizeof(mylist)=184
i=14, len(mylist)=14, sys.getsizeof(mylist)=184
i=15, len(mylist)=15, sys.getsizeof(mylist)=184
i=16, len(mylist)=16, sys.getsizeof(mylist)=184
i=17, len(mylist)=17, sys.getsizeof(mylist)=248
i=18, len(mylist)=18, sys.getsizeof(mylist)=248
i=19, len(mylist)=19, sys.getsizeof(mylist)=248
i=20, len(mylist)=20, sys.getsizeof(mylist)=248
i=21, len(mylist)=21, sys.getsizeof(mylist)=248
i=22, len(mylist)=22, sys.getsizeof(mylist)=248
i=23, len(mylist)=23, sys.getsizeof(mylist)=248
i=24, len(mylist)=24, sys.getsizeof(mylist)=248
i=25, len(mylist)=25, sys.getsizeof(mylist)=312
i=26, len(mylist)=26, sys.getsizeof(mylist)=312
i=27, len(mylist)=27, sys.getsizeof(mylist)=312
i=28, len(mylist)=28, sys.getsizeof(mylist)=312
i=29, len(mylist)=29, sys.getsizeof(mylist)=312
```

[41]:
```
t = (10, 20, 30, 40, 50)
sys.getsizeof(t)
```

[41]: 80

[42]:
```
person = ('Reuven', 'Lerner', 46)
```

[43]:
```
person[0]
```

[43]:
```
'Reuven'
```

[44]:
```
person[1]
```

[44]:
```
'Lerner'
```

[45]:
```
person[2]
```

[45]:
```
46
```

[46]:
```
# namedtuple

from collections import namedtuple
```

[47]:
```
Person = namedtuple('Person', ['first', 'last', 'shoesize'])
```

[48]:
```
type(Person)
```

[48]:
```
type
```

[49]:
```
Person.__bases__
```

[49]:
```
(tuple,)
```

[50]:
```
str.__name__
```

[50]:
```
'str'
```

[51]:
```
list.__name__
```

[51]:
```
'list'
```

[52]:
```
Person.__name__
```

[52]:
```
'Person'
```

[53]:
```
Person.__name__ = 'otherthing'
```

[54]:
```
Person.__name__
```

[54]:
```
'otherthing'
```

[55]:
```
Person = namedtuple('Dog', ['first', 'last', 'shoesize'])
```

[56]:
```
Person.__name__
```

[56]:
```
'Dog'
```

[57]:
```
p = Person('Reuven', 'Lerner', 46)
```

[58]:
```
p[0]
```

[58]:
```
'Reuven'
```

[59]:
```
p[1]
```

[59]:
```
'Lerner'
```

[60]:
```
p[2]
```

[60]:
```
46
```

[61]:
```
p.first
```

[61]:
```
'Reuven'
```

[62]:
```
p.last
```

[62]:
```
'Lerner'
```

[63]:
```
p.shoesize
```

[63]:
```
46
```

[64]:
```
p
```

[64]:
```
Dog(first='Reuven', last='Lerner', shoesize=46)
```

[65]:
```
Person = namedtuple('Person', ['first', 'last', 'shoesize'])
```

[66]:
```
p = Person('Reuven', 'Lerner', 46)
```

[67]:
```
p
```

[67]:
```
Person(first='Reuven', last='Lerner', shoesize=46)
```

[68]:
```
# _replace returns a new tuple
#
p._replace(first='OtherName')
```

[68]:
```
Person(first='OtherName', last='Lerner', shoesize=46)
```

3 Exercise: Bookstore

1. Define a `Book` class using `namedtuple` with fields `title`, `price`, `author`.
2. Define a list, `inventory`, with 4 instances of `Book`.
3. Ask the user, repeatedly, what book they want to buy.
   - Empty string? Stop asking
   - A book that is in `inventory`? Print all details, and add price to `total`

- Not there? Scold the user a bit

4. In the end, tell the user the total bill.

[3]:
```
from collections import namedtuple

Book = namedtuple('Book', ['title', 'price', 'author'])

# Book = namedtuple('Book', 'title price author')

b1 = Book('title1', 100, 'author1')
b2 = Book('title2', 75, 'author2')
b3 = Book('title3', 150, 'author1')
b4 = Book('title4', 130, 'author2')

inventory = [b1, b2, b3, b4]
total = 0

while True:
    to_buy = input('Enter book: ').strip()

    # if to_buy == '':   # not Pythonic
    #     break

    if not to_buy:
        break

    found_book = False
    for one_book in inventory:
        if one_book.title == to_buy:
            total += one_book.price
            print(f'{to_buy} by {one_book.author} costs {one_book.price}; total␣
 ↪is now {total}')
            found_book = True

    if not found_book:
        print(f'Did not find {to_buy}')

print(total)
```

```
Enter book:  title1
title1 by author1 costs 100; total is now 100
Enter book:  tiel123512
Did not find tiel123512
Enter book:
100
```

[4]:
```
# version using for-else

from collections import namedtuple

Book = namedtuple('Book', ['title', 'price', 'author'])

# Book = namedtuple('Book', 'title price author')

b1 = Book('title1', 100, 'author1')
b2 = Book('title2', 75, 'author2')
b3 = Book('title3', 150, 'author1')
b4 = Book('title4', 130, 'author2')

inventory = [b1, b2, b3, b4]
total = 0

while True:
    to_buy = input('Enter book: ').strip()

    if not to_buy:
        break

    found_book = False
    for one_book in inventory:
        if one_book.title == to_buy:
            total += one_book.price
            print(f'{to_buy} by {one_book.author} costs {one_book.price}; total␣
 ↪is now {total}')
            break

    else:
        print(f'Did not find {to_buy}')

print(total)
```

```
Enter book:  title1
title1 by author1 costs 100; total is now 100
Enter book:  title2
title2 by author2 costs 75; total is now 175
Enter book:  titleq3145325
Did not find titleq3145325
Enter book:
175
```

[7]:
```
from collections import namedtuple

Book = namedtuple('Book', ['title', 'price', 'author'])

# Book = namedtuple('Book', 'title price author')

b1 = Book('title1', 100, 'author1')
b2 = Book('title2', 75, 'author2')
b3 = Book('title3', 150, 'author1')
b4 = Book('title4', 130, 'author2')

inventory = [b1, b2, b3, b4]
total = 0

# := is officially assignment expression operator
# walrus

while to_buy := input('Enter book: ').strip():

    for one_book in inventory:
        if one_book.title == to_buy:
            total += one_book.price
            print(f'{to_buy} by {one_book.author} costs {one_book.price}; total␣
 ↪is now {total}')
            break

    else:
        print(f'Did not find {to_buy}')

print(total)
```

```
Enter book:  title1
title1 by author1 costs 100; total is now 100
Enter book:  titleasdfa
Did not find titleasdfa
Enter book:
100
```

[6]:
```
x := 5
```

```
  Cell In[6], line 1
    x := 5
      ^
SyntaxError: invalid syntax
```

[8]:
```
x = 10
y = 20

print(f'{x} + {y} = {x+y}')
```

```
10 + 20 = 30
```

[9]:
```
print('{x} + {y} = {x+y}')
```

```
{x} + {y} = {x+y}
```

4 Dictionaries

1. Hash table
2. Every key has a value, every value has a key
3. Every key has to be hashable (more or less immutable)
4. Lookup of a key is O(1) + epsilon

[10]:
```
d = {'a':10, 'b':20, 'c':30}

# search the keys with "in"
'c' in d
```

[10]:
```
True
```

[11]:
```
# don't use dict.keys for searching/looping
'c' in d.keys()
```

[11]:
```
True
```

[12]:
```
for key in d.keys():
    print(f'{key}: {d[key]}')
```

```
a: 10
b: 20
c: 30
```

[15]:
```
# we can iterate on the dict, and get the keys

for key in d:
    print(f'{key}: {d[key]}')
```

```
a: 10
b: 20
c: 30
```

[16]:
```
list(d)
```

[16]:
```
['a', 'b', 'c']
```

[18]:
```
# we can use dict.values

30 in d.values()      # search is O(n)
```

[18]:
```
True
```

[19]:
```
# dict.items() is a better way to iterate over the keys + values

for t in d.items():
    print(t)
```

```
('a', 10)
('b', 20)
('c', 30)
```

[21]:
```
# tuple unpacking

for t in d.items():
    key, value = t      # two variables = two values
    print(f'{key}: {value}')
```

```
a: 10
b: 20
c: 30
```

[22]:
```
# tuple unpacking in the for loop

for key, value in d.items():
    print(f'{key}: {value}')
```

```
a: 10
b: 20
c: 30
```

[23]:
```
# unpacking

mylist = [10, 20, 30, 40, 50]

v,w,x,y,z = mylist
```

[24]:
```
v
```

[24]:
```
10
```

[25]:
```
w
```

[25]:
```
20
```

[26]:
```
z
```

[26]: 50

[27]:
```
x,y = mylist
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[27], line 1
----> 1 x,y = mylist

ValueError: too many values to unpack (expected 2)
```

[28]:
```
a,b,c,d,e,f = mylist
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[28], line 1
----> 1 a,b,c,d,e,f = mylist

ValueError: not enough values to unpack (expected 6, got 5)
```

[29]:
```
x, *y, z = mylist
```

[30]:
```
x
```

[30]: 10

[31]:
```
z
```

[31]: 50

[32]:
```
y
```

[32]: [20, 30, 40]

[33]:
```
w,x,*y, z = mylist
```

[34]:
```
w
```

[34]: 10

[35]:
```
x
```

[35]: 20

[36]:
```
y
```

[36]:
```
[30, 40]
```

[37]:
```
z
```

[37]:
```
50
```

[38]:
```
*w,x,y, z = mylist
```

[39]:
```
w
```

[39]:
```
[10, 20]
```

[40]:
```
x
```

[40]:
```
30
```

[41]:
```
hash('a')
```

[41]:
```
8450858394419656295
```

[42]:
```
hash('b')
```

[42]:
```
-9047225283332092375
```

5 Old-style dicts

1. Started with 8 locations in memory
2. When we want to store in the dict, Python runs `hash(key) % len(d)`
3. When we get to 2/3 fullness, we double the dict size
4. Collision – jump to next place, then all around via a function

6 New-style dicts

1. Two data structures behind the scenes
   - A table that starts with zero lines, and grows – index (starting at 0), key, value
   - A C array that starts with 8 locations
2. If we want to write to the dict, we write the new key-value pair to the table. We calculate `hash(key) % len(array)`, and write to that location in the array the index of the table.
3. To retrieve from our dict:
   - `hash(key) % len(array)` gives us the array index
   - The item in the array tells us where to grab the value from the table

[43]:
```
d = {'a':10, 'b':20, 'c':30}

d['a']
```

[43]:
```
10
```

[44]:
```
d['x']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[44], line 1
----> 1 d['x']

KeyError: 'x'
```

[45]:
```
# what if I want to avoid the error?

d.get('x')     # get None back if it doesn't exist
```

[46]:
```
d.get('x', 'Sorry, no x here!')
```

[46]:
```
'Sorry, no x here!'
```

[47]:
```
d
```

[47]:
```
{'a': 10, 'b': 20, 'c': 30}
```

[48]:
```
# I want to assign this pair to the dict, but only if 'c' doesn't exist as a key
# d['c'] = 999

d.setdefault('c', 999)
```

[48]:
```
30
```

[49]:
```
d
```

[49]:
```
{'a': 10, 'b': 20, 'c': 30}
```

[50]:
```
d.setdefault('x', 888)
d
```

[50]:
```
{'a': 10, 'b': 20, 'c': 30, 'x': 888}
```

[51]:
```
# set up a new dict with known keys and all values being 0

d = dict.fromkeys('xyz', 0)
d
```

[51]:
```
{'x': 0, 'y': 0, 'z': 0}
```

[54]:
```
d = dict.fromkeys('this that the_other'.split(), 0)
d
```

[54]:
```
{'this': 0, 'that': 0, 'the_other': 0}
```

[55]:
```
d = dict.fromkeys('xyz', [])
d
```

[55]:
```
{'x': [], 'y': [], 'z': []}
```

[56]:
```
d['x'].append(10)
d['y'].append(20)

d
```

[56]:
```
{'x': [10, 20], 'y': [10, 20], 'z': [10, 20]}
```

[57]:
```
x = y = 10

x
```

[57]:
```
10
```

[58]:
```
y
```

[58]:
```
10
```

[59]:
```
x = y = []

x.append(10)
y.append(20)

x
```

[59]:
```
[10, 20]
```

[60]:
```
d1 = {'a':10, 'b':20, 'c':30}
d2 = {'c':30, 'b':20, 'a':10}
```

[61]:
```
d1 == d2
```

[61]:
```
True
```

[62]:
```
d = {}
d['a'] = 100
d['x'] = 200
d['b'] = 300
d['d'] = 400
```

```
d
```

[62]:
```
{'a': 100, 'x': 200, 'b': 300, 'd': 400}
```

[63]:
```
list(d)
```

[63]:
```
['a', 'x', 'b', 'd']
```

[64]:
```
for key, value in d.items():
    print(f'{key}: {value}')
```

```
a: 100
x: 200
b: 300
d: 400
```

[65]:
```
d['a'] = 999
list(d)
```

[65]:
```
['a', 'x', 'b', 'd']
```

[66]:
```
d.pop('a')
```

[66]:
```
999
```

[67]:
```
d['a'] = 777
d
```

[67]:
```
{'x': 200, 'b': 300, 'd': 400, 'a': 777}
```

[68]:
```
from collections import defaultdict
```

[69]:
```
# what about a default dict of 0?

d = defaultdict(0)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[69], line 3
      1 # what about a default dict of 0?
----> 3 d = defaultdict(0)

TypeError: first argument must be callable or None
```

[70]:
```
# None means: Regular dictionary
# callable: any function or class, which takes 0 arguments
```

```
# the result from the callable is assigned, with the key, to the dict

int()
```

[70]: 0

[71]:
```
d = defaultdict(int)   # default is int(), or 0
```

[72]:
```
d['a']
```

[72]: 0

[73]:
```
d['b'] += 15
```

[74]:
```
d
```

[74]: defaultdict(int, {'a': 0, 'b': 15})

[75]:
```
d['c'] += 1
d
```

[75]: defaultdict(int, {'a': 0, 'b': 15, 'c': 1})

[76]:
```
d = defaultdict(dict)
```

[77]:
```
d['a']['b'] = 100
```

[78]:
```
d['c']['d'] = 200
d
```

[78]: defaultdict(dict, {'a': {'b': 100}, 'c': {'d': 200}})

[79]:
```
import time

d = defaultdict(time.time)
```

[80]:
```
d['a']   # if we ask for a new key, time.time() runs, its value is stored
```

[80]: 1707907802.226869

[81]:
```
d['b']
```

[81]: 1707907803.9616299

[82]:
```
d['c']
```

[82]: 1707907806.761536

7 Exercise: Travel

1. We'll create a dict in which the keys are strings (country names) and the values are lists (cities ij those countries).
2. Ask the user to enter `city, country` for somewhere they've traveled.
   - Give an error message if you don't get the right format
3. Add to the dictionary's `country` key the value of `city`.
4. When the user gives us an empty string, print each country – and in each country, print each city.

Example:

```
Where have you traveled: Beijing, China
Where have you traveled: Shanghai, China
Where have you traveled: Chicago, USA
Where have you traveled: Boston, USA
Where have you traveled: [ENTER]
```

Dict will be

```
{'China': ['Beijing', 'Shanghai'],
 'USA': ['Chicago', 'Bostona']}
```

[83]:
```
from collections import defaultdict
```

[84]:
```
list()
```

[84]:
```
[]
```

[85]:
```
mylist = [10, 20, 30]
mylist()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[85], line 2
      1 mylist = [10, 20, 30]
----> 2 mylist()

TypeError: 'list' object is not callable
```

[88]:
```
from collections import defaultdict

all_places = defaultdict(list)

while True:
    print(f'{all_places=}')
    one_place = input('Enter a place: ').strip()

    if not one_place:
```

```
        break

    if ',' not in one_place:
        print(f'Bad format!')
        continue

    one_city, one_country = one_place.split(',')
    all_places[one_country.strip()].append(one_city.strip())
```

```
all_places=defaultdict(<class 'list'>, {})

Enter a place:  Beijing, China

all_places=defaultdict(<class 'list'>, {'China': ['Beijing']})

Enter a place:  Chicago, USA

all_places=defaultdict(<class 'list'>, {'China': ['Beijing'], 'USA':
['Chicago']})

Enter a place:  London, UK

all_places=defaultdict(<class 'list'>, {'China': ['Beijing'], 'USA':
['Chicago'], 'UK': ['London']})

Enter a place:  Shanghai, China

all_places=defaultdict(<class 'list'>, {'China': ['Beijing', 'Shanghai'], 'USA':
['Chicago'], 'UK': ['London']})

Enter a place:  asdfsafafda

Bad format!
all_places=defaultdict(<class 'list'>, {'China': ['Beijing', 'Shanghai'], 'USA':
['Chicago'], 'UK': ['London']})

Enter a place:
```

[89]:
```
all_places
```

[89]:
```
defaultdict(list,
            {'China': ['Beijing', 'Shanghai'],
             'USA': ['Chicago'],
             'UK': ['London']})
```

[91]:
```
for country, all_cities in all_places.items():
    print(country)
    for one_city in all_cities:
        print(f'\t{one_city}')
```

```
China
        Beijing
        Shanghai
USA
```

```
        Chicago
UK
        London
```

[92]:
```
# Counter

from collections import Counter
```

[93]:
```
# we can think of Counter as defaultdict(int)

c = Counter()
c['a'] += 1
c['b'] += 5
c
```

[93]:
```
Counter({'b': 5, 'a': 1})
```

[94]:
```
#  don't do this!

# instead, create Counter with an iterable of hashables

c = Counter([10, 20, 30, 40, 20, 30, 40, 20, 30])
c
```

[94]:
```
Counter({20: 3, 30: 3, 40: 2, 10: 1})
```

[95]:
```
list(c)
```

[95]:
```
[10, 20, 30, 40]
```

[96]:
```
c.most_common()
```

[96]:
```
[(20, 3), (30, 3), (40, 2), (10, 1)]
```

[97]:
```
c.most_common(2)
```

[97]:
```
[(20, 3), (30, 3)]
```

[98]:
```
c2 = Counter([20, 30, 40])
c | c2
```

[98]:
```
Counter({20: 3, 30: 3, 40: 2, 10: 1})
```

[99]:
```
c
```

[99]:
```
Counter({20: 3, 30: 3, 40: 2, 10: 1})
```

[100]:
```
c & c2
```

[100]:
```
Counter({20: 1, 30: 1, 40: 1})
```

[101]:
```
c + c2
```

[101]:
```
Counter({20: 4, 30: 4, 40: 3, 10: 1})
```

[102]:
```
c = Counter(open('/Users/reuven/Courses/Current/data/alice-in-wonderland.txt').
↪read())
```

[103]:
```
c
```

[103]:
```
Counter({' ': 12387,
         'e': 6671,
         't': 5152,
         'o': 4242,
         'a': 4058,
         'i': 3547,
         'n': 3530,
         'h': 3122,
         'r': 3111,
         's': 3057,
         'd': 2245,
         'l': 2221,
         '\n': 1702,
         'u': 1634,
         'c': 1435,
         'g': 1230,
         'w': 1165,
         'm': 1064,
         'f': 1061,
         'y': 985,
         ',': 927,
         'p': 849,
         'b': 742,
         '"': 710,
         '.': 621,
         'k': 509,
         'v': 389,
         'I': 375,
         'A': 352,
         '-': 287,
         "'": 260,
         'T': 241,
         'E': 187,
         '!': 176,
         'S': 159,
         '_': 150,
```

```
'P': 140,
'R': 133,
'j': 129,
'G': 122,
'D': 120,
'O': 116,
'N': 113,
'W': 103,
'C': 100,
'F': 92,
'H': 92,
'L': 90,
'M': 79,
';': 78,
'?': 70,
'x': 70,
'1': 64,
'B': 62,
'q': 62,
'Y': 58,
'U': 53,
'K': 42,
':': 40,
'z': 34,
'Q': 34,
'*': 33,
'(': 33,
')': 33,
'/': 31,
'[': 27,
']': 27,
'0': 24,
'3': 21,
'V': 20,
'9': 15,
'J': 14,
'5': 12,
'8': 11,
'2': 10,
'6': 9,
'4': 9,
'7': 6,
'X': 4,
'&': 2,
'@': 2,
'$': 2,
'\ufeff': 1,
```

```
             '#': 1,
             'ù': 1,
             '%': 1})
```

[104]:
```
c.most_common(10)
```

[104]:
```
[(' ', 12387),
 ('e', 6671),
 ('t', 5152),
 ('o', 4242),
 ('a', 4058),
 ('i', 3547),
 ('n', 3530),
 ('h', 3122),
 ('r', 3111),
 ('s', 3057)]
```

8 Functions

[105]:
```
s = 'abcd'
x = len(s)

type(x)
```

[105]:
```
int
```

[106]:
```
x
```

[106]:
```
4
```

[107]:
```
x = s.upper()
type(x)
```

[107]:
```
str
```

[108]:
```
x
```

[108]:
```
'ABCD'
```

[109]:
```
x = s.upper
```

[110]:
```
x
```

[110]:
```
<function str.upper()>
```

[111]:
```
type(x)
```

[111]:
```
builtin_function_or_method
```

[112]:
```
x()
```

[112]:
```
'ABCD'
```

[113]:
```
d = {'a':10, 'b':20, 'c':30}

for key, value in d.items():
    print(f'{key}: {value}')
```

```
a: 10
b: 20
c: 30
```

[114]:
```
# this fails
for key, value in d.items:
    print(f'{key}: {value}')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[114], line 2
      1 # this fails
----> 2 for key, value in d.items:
      3     print(f'{key}: {value}')

TypeError: 'builtin_function_or_method' object is not iterable
```

[116]:
```
# When I use def, two things happen:
# (1) I create a function object
# (2) I assign the function object to the variable myfunc

def myfunc():
    return 5

type(myfunc)
```

[116]:
```
function
```

[117]:
```
myfunc()
```

[117]:
```
5
```

[118]:
```
def myfunc(x):
    return f'Hello, {x}'

myfunc(10)
```

[118]:
```
'Hello, 10'
```

[119]:
```
# no arguments
myfunc()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[119], line 2
      1 # no arguments
----> 2 myfunc()

TypeError: myfunc() missing 1 required positional argument: 'x'
```

[120]:
```
# function object

def myfunc():
    print("Hello!")
```

[121]:
```
type(myfunc)
```

[121]:
```
function
```

[122]:
```
# attributes - names that come after a .

myfunc.__code__     # "dunder code" -- "double underscore before and after code"
```

[122]:
```
<code object myfunc at 0x10e9d3590, file
"/var/folders/d9/v8tsklln4477fll05wkgcpth0000gn/T/ipykernel_1402/4225771070.py",
line 3>
```

[123]:
```
dir(myfunc.__code__)
```

[123]:
```
['__class__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__ne__',
```

```
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '_co_code_adaptive',
 '_varname_from_oparg',
 'co_argcount',
 'co_cellvars',
 'co_code',
 'co_consts',
 'co_exceptiontable',
 'co_filename',
 'co_firstlineno',
 'co_flags',
 'co_freevars',
 'co_kwonlyargcount',
 'co_lines',
 'co_linetable',
 'co_lnotab',
 'co_name',
 'co_names',
 'co_nlocals',
 'co_positions',
 'co_posonlyargcount',
 'co_qualname',
 'co_stacksize',
 'co_varnames',
 'replace']
```

[124]:
```
myfunc.__code__.co_varnames
```

[124]:
```
()
```

[125]:
```
# byte codes!

myfunc.__code__.co_code
```

[125]:
```
b'\x97\x00t\x01\x00\x00\x00\x00\x00\x00\x00d\x01\xab\x01\x00\x00\x00\x00\x00
\x00\x01\x00y\x00'
```

[126]:
```
import dis
```

[127]:
```
dis.dis(myfunc)
```

```
  3           0 RESUME                   0

  4           2 LOAD_GLOBAL              1 (NULL + print)
             12 LOAD_CONST               1 ('Hello!')
             14 CALL                     1
             22 POP_TOP
             24 RETURN_CONST             0 (None)
```

[128]:
```
myfunc()
```

```
Hello!
```

[129]:
```
myfunc.__code__.co_consts
```

[129]:
```
(None, 'Hello!')
```

[130]:
```
def hello(name):
    return f'Hello, {name}!'
```

[131]:
```
dis.dis(hello)
```

```
  1           0 RESUME                   0

  2           2 LOAD_CONST               1 ('Hello, ')
              4 LOAD_FAST                0 (name)
              6 FORMAT_VALUE             0
              8 LOAD_CONST               2 ('!')
             10 BUILD_STRING             3
             12 RETURN_VALUE
```

[132]:
```
hello('world')
```

[132]:
```
'Hello, world!'
```

[133]:
```
hello()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[133], line 1
----> 1 hello()

TypeError: hello() missing 1 required positional argument: 'name'
```

[134]:
```
# positional argument

# parameter: name
# argument: 'world'
```

```
hello('world')
```

[134]: 'Hello, world!'

[135]:
```
def hello(first, last):
    return f'Hello, {first} {last}'

# parameter: first last
# argument:   'a'   'b'

hello('a', 'b')
```

[135]: 'Hello, a b'

[137]:
```
# keyword arguments
# these always look like: name=value

# parameters:  first    last
# arguments:   'a'       'b'

hello(first='a', last='b')
```

[137]: 'Hello, a b'

[138]:
```
hello(last='a', first='b')
```

[138]: 'Hello, b a'

[140]:
```
# all positional arguments must come before all keyword arguments

# parameters: first  last
# arguments:   'a'    'b'

hello('a', last='b')
```

[140]: 'Hello, a b'

[141]:
```
hello(first='a', 'b')
```

```
  Cell In[141], line 1
    hello(first='a', 'b')
                     ^
SyntaxError: positional argument follows keyword argument
```

[142]:
```
def hello():
    print('Hello!')
```

[143]:
```
hello('world')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[143], line 1
----> 1 hello('world')

TypeError: hello() takes 0 positional arguments but 1 was given
```

[144]:
```
hello.__code__.co_argcount
```

[144]:
```
0
```

[145]:
```
def hello(name):
    return f'Hello, {name}!'
```

[146]:
```
hello()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[146], line 1
----> 1 hello()

TypeError: hello() missing 1 required positional argument: 'name'
```

[147]:
```
hello.__code__.co_argcount
```

[147]:
```
1
```

[148]:
```
hello.__code__.co_varnames
```

[148]:
```
('name',)
```

[149]:
```
# regular parameter -- can take either positional or keyword args
def hello(name):
    return f'Hello, {name}!'
```

[150]:
```
hello('a')
```

[150]:
```
'Hello, a!'
```

[151]:
```
hello(name='a')
```

[151]:
```
'Hello, a!'
```

[152]:
```
hello(5)
```

[152]:
```
'Hello, 5!'
```

[153]:
```
hello([10, 20, 30])
```

[153]:
```
'Hello, [10, 20, 30]!'
```

[154]:
```
hello(hello)
```

[154]:
```
'Hello, <function hello at 0x10e9e45e0>!'
```

[155]:
```
# parameter type #2: regular parameter with a default argument value

def hello(name='whoever'):
    return f'Hello, {name}!'
```

[157]:
```
# parameters: name
# arguments: 'world'

hello('world')
```

[157]:
```
'Hello, world!'
```

[160]:
```
# parameters: name
# arguments:  'whoever'

hello()
```

[160]:
```
'Hello, whoever!'
```

[158]:
```
hello.__code__.co_argcount
```

[158]:
```
1
```

[159]:
```
hello.__defaults__
```

[159]:
```
('whoever',)
```

[161]:
```
def add(first=10, second=5):
    return first + second
```

[162]:
```
add.__defaults__
```

[162]:
```
(10, 5)
```

[163]:
```
# parameters: first  second
# arguments:   10      5

add()
```

[163]:
```
15
```

[164]:
```
# parameters: first  second
# arguments:    10       7

add(second=7)
```

[164]:
```
17
```

[165]:
```
add.__defaults__ = (100, 200)
```

[166]:
```
add()
```

[166]:
```
300
```

[169]:
```
def add_one(x):
    x.append(1)
    return x

mylist = [10, 20, 30]
print(id(mylist))
add_one(mylist)
print(id(mylist))
```

```
4541295552
4541295552
```

[168]:
```
mylist
```

[168]:
```
[10, 20, 30, 1]
```

[170]:
```
def add_one(x=[]):
    x.append(1)
    return x

add_one(mylist)
```

[170]:
```
[10, 20, 30, 1, 1]
```

[171]:
```
add_one(mylist)
```

[171]:
```
[10, 20, 30, 1, 1, 1]
```

[172]:
```
mylist
```

[172]:
```
[10, 20, 30, 1, 1, 1]
```

[173]:
```
add_one()
```

[173]:
```
[1]
```

[174]:
```
add_one()
```

[174]:
```
[1, 1]
```

[175]:
```
add_one()
```

[175]:
```
[1, 1, 1]
```

[176]:
```
def add_one(x=[]):
    x.append(1)
    return x

add_one.__defaults__
```

[176]:
```
([],)
```

[177]:
```
# parameters:   x
# arguments:   []

add_one()
```

[177]:
```
[1]
```

[179]:
```
add_one.__defaults__
```

[179]:
```
([1],)
```

[180]:
```
def add_one(x=None):
    if x is None:
        x = []

    x.append(1)
    return x
```

9 Parameter types

1. Mandatory parameters – either positional or keyword
2. Optional parameters (with default) – either positional or keyword

[184]:
```
def myfunc(a=1, b):
    pass   # do nothing, but let the program run
```

```
  Cell In[184], line 1
    def myfunc(a=1, b):
                    ^
```

```
SyntaxError: parameter without a default follows parameter with a default
```

[185]:
```
s = 'a b c d e'
s.split(' ')
```

[185]:
```
['a', 'b', 'c', 'd', 'e']
```

[186]:
```
s.split()
```

[186]:
```
['a', 'b', 'c', 'd', 'e']
```

[187]:
```
help(str.split)
```

```
Help on method_descriptor:

split(self, /, sep=None, maxsplit=-1)
    Return a list of the substrings in the string, using sep as the separator
string.

      sep
        The separator used to split the string.

        When set to None (the default value), will split on any whitespace
        character (including \n \r \t \f and spaces) and will discard
        empty strings from the result.
      maxsplit
        Maximum number of splits (starting from the left).
        -1 (the default value) means no limit.

    Note, str.split() is mainly useful for data that has been intentionally
    delimited.  With natural text that includes punctuation, consider using
    the regular expression module.
```

[188]:
```
s.split(maxsplit=3)
```

[188]:
```
['a', 'b', 'c', 'd e']
```

[189]:
```
def mysum(numbers):
    total = 0

    for one_number in numbers:
        total += one_number

    return total
```

[190]:
```
mysum([10, 20, 30])
```

[190]: 60

[191]:
```
# right now, numbers has to be a list/tuple of numbers
# what if I want to get separate numeric arguments?

mysum(10, 20, 30)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[191], line 4
      1 # right now, numbers has to be a list/tuple of numbers
      2 # what if I want to get separate numeric arguments?
----> 4 mysum(10, 20, 30)

TypeError: mysum() takes 1 positional argument but 3 were given
```

[192]:
```
def mysum(a=0, b=0, c=0, d=0, e=0, f=0, g=0):
    return a + b + c + d + e + f + g
```

[193]:
```
# We can define a parameter named *args
# the parameter can be called anything (but args is traditional)
# args will be a tuple

# its values will be all of the positional arguments that weren't grabbed by␣
 ↪other parameters

def mysum(*numbers):   # splat
    total = 0
    for one_number in numbers:
        total += one_number
    return total

mysum(10, 20, 30)
```

[193]: 60

[194]:
```
mysum(a=10, b=20)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[194], line 1
----> 1 mysum(a=10, b=20)

TypeError: mysum() got an unexpected keyword argument 'a'
```

[195]:
```
mysum(numbers=[10, 20, 30])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[195], line 1
----> 1 mysum(numbers=[10, 20, 30])

TypeError: mysum() got an unexpected keyword argument 'numbers'
```

[198]:
```
def myfunc(a, b, *args):
    return f'{a=}, {b=}, {args=}'
```

[199]:
```
myfunc(10, 20, 30, 40, 50)
```

[199]:
```
'a=10, b=20, args=(30, 40, 50)'
```

[200]:
```
myfunc(10, 20)
```

[200]:
```
'a=10, b=20, args=()'
```

[201]:
```
print('a', 'b', 'c', 'd')
```

```
a b c d
```

[202]:
```
print('a', 'b', 'c', 'd', sep='*')
```

```
a*b*c*d
```

[203]:
```
help(print)
```

```
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current sys.stdout.
    flush
      whether to forcibly flush the stream.
```

[ ]:
```

```

10 Parameter types

1. Mandatory parameters – either positional or keyword
2. Optional parameters (with default) – either positional or keyword
3. `*args`, with all unclaimed positional arguments

11 Exercise: `all_lines`

Write a function, `all_lines`, that takes: - `outfilename`, a string with the name of the file to which we will write - One or more additional filenames, that we'll take for our input

All of the lines from the input files (in order) should be printed to the output file.

If you want, number the files and lines.

[ ]:
```
def all_lines(outfilename, *infilenames):
    f = open(outfilename, 'w')
    for one_infilename in infilenames:
        for one_line in open(one_infilename):
            f.write(one_line)
    f.close()
```

[204]:
```
# improved, more secure version 2.0

def all_lines(outfilename, *infilenames):
    with open(outfilename, 'w') as f:
        # f.__enter__()
        for one_infilename in infilenames:
            with open(one_infilename) as infile:
                for one_line in infile:
                    f.write(one_line)
            # infile.__exit__() -- close
        # f.__exit__()  -- flush + close
```

[220]:
```
for i in range(5):
    with open(f'junkfile-{i}.txt', 'w') as f:
        for j in range(3):
            f.write(f'junkfile {i}: {j} abcd\n')
```

[221]:
```
!ls *.txt
```

```
junkfile-0.txt  junkfile-2.txt  junkfile-4.txt
junkfile-1.txt  junkfile-3.txt  outfile.txt
```

[222]:
```
!cat junkfile-0.txt
```

```
junkfile 0: 0 abcd
junkfile 0: 1 abcd
junkfile 0: 2 abcd
```

[223]:
```
all_lines('outfile.txt', 'junkfile-0.txt', 'junkfile-1.txt', 'junkfile-2.txt')
```

[224]:
```
!cat outfile.txt
```

```
junkfile 0: 0 abcd
junkfile 0: 1 abcd
junkfile 0: 2 abcd
junkfile 1: 0 abcd
junkfile 1: 1 abcd
junkfile 1: 2 abcd
junkfile 2: 0 abcd
junkfile 2: 1 abcd
junkfile 2: 2 abcd
```

[225]:
```
# version 3.0 -- with numbering!

def all_lines(outfilename, *infilenames):
    with open(outfilename, 'w') as f:
        # f.__enter__()
        for file_index, one_infilename in enumerate(infilenames):
            with open(one_infilename) as infile:
                for line_index, one_line in enumerate(infile):
                    f.write(f'{file_index} {line_index} {one_line}')
            # infile.__exit__() -- close
        # f.__exit__()  -- flush + close
```

[226]:
```
all_lines('outfile.txt', 'junkfile-0.txt', 'junkfile-1.txt', 'junkfile-2.txt')
```

[227]:
```
!cat outfile.txt
```

```
0 0 junkfile 0: 0 abcd
0 1 junkfile 0: 1 abcd
0 2 junkfile 0: 2 abcd
1 0 junkfile 1: 0 abcd
1 1 junkfile 1: 1 abcd
1 2 junkfile 1: 2 abcd
2 0 junkfile 2: 0 abcd
2 1 junkfile 2: 1 abcd
2 2 junkfile 2: 2 abcd
```

[228]:
```
!ls junkfile*
```

```
junkfile-0.txt  junkfile-1.txt  junkfile-2.txt  junkfile-3.txt  junkfile-4.txt
```

[233]:
```
import glob
glob.glob('junkfile*')
```

[233]:
```
['junkfile-4.txt',
 'junkfile-3.txt',
```

```
 'junkfile-2.txt',
 'junkfile-0.txt',
 'junkfile-1.txt']
```

[234]:
```
all_lines('outfile.txt',
          *glob.glob('junkfile*'))
```

[232]:
```
!cat outfile.txt
```

```
0 0 junkfile 4: 0 abcd
0 1 junkfile 4: 1 abcd
0 2 junkfile 4: 2 abcd
1 0 junkfile 3: 0 abcd
1 1 junkfile 3: 1 abcd
1 2 junkfile 3: 2 abcd
2 0 junkfile 2: 0 abcd
2 1 junkfile 2: 1 abcd
2 2 junkfile 2: 2 abcd
3 0 junkfile 0: 0 abcd
3 1 junkfile 0: 1 abcd
3 2 junkfile 0: 2 abcd
4 0 junkfile 1: 0 abcd
4 1 junkfile 1: 1 abcd
4 2 junkfile 1: 2 abcd
```

[243]:
```
def myfunc(*args):
    for index, one_item in enumerate(args):
        print(f'{index}: {one_item}')

myfunc(*open('outfile.txt'))
```

```
0: 0 0 junkfile 4: 0 abcd

1: 0 1 junkfile 4: 1 abcd

2: 0 2 junkfile 4: 2 abcd

3: 1 0 junkfile 3: 0 abcd

4: 1 1 junkfile 3: 1 abcd

5: 1 2 junkfile 3: 2 abcd

6: 2 0 junkfile 2: 0 abcd

7: 2 1 junkfile 2: 1 abcd

8: 2 2 junkfile 2: 2 abcd
```

```
9: 3 0 junkfile 0: 0 abcd

10: 3 1 junkfile 0: 1 abcd

11: 3 2 junkfile 0: 2 abcd

12: 4 0 junkfile 1: 0 abcd

13: 4 1 junkfile 1: 1 abcd

14: 4 2 junkfile 1: 2 abcd
```

[ ]:

12 Parameter types

1. Mandatory parameters – either positional or keyword
2. Optional parameters (with default) – either positional or keyword
3. `*args`, with all unclaimed positional arguments

[244]:
```
# **kwargs ("double splat")
# kwargs is a dict containing all keyword args that no one else grabbed

def myfunc(x, **kwargs):
    print(f'{x=}, {kwargs=}')
```

[245]:
```
myfunc(10, 20)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[245], line 1
----> 1 myfunc(10, 20)

TypeError: myfunc() takes 1 positional argument but 2 were given
```

[246]:
```
myfunc(10, a=100, b=200, c=300)
```

```
x=10, kwargs={'a': 100, 'b': 200, 'c': 300}
```

[247]:
```
myfunc(10, 5=100)
```

```
  Cell In[247], line 1
    myfunc(10, 5=100)
               ^
```

```
SyntaxError: expression cannot contain assignment, perhaps you meant "=="?
```

[248]:
```
myfunc(10, {5:100})
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[248], line 1
----> 1 myfunc(10, {5:100})

TypeError: myfunc() takes 1 positional argument but 2 were given
```

[249]:
```
myfunc(x=100, y=200, z=300)
```

```
x=100, kwargs={'y': 200, 'z': 300}
```

[250]:
```
# parameters: x
# arguments: 100/200?

myfunc(100, x=200)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[250], line 1
----> 1 myfunc(100, x=200)

TypeError: myfunc() got multiple values for argument 'x'
```

[251]:
```
info = {'name':'Reuven', 'shoesize':46}

myfunc(100, info)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[251], line 3
      1 info = {'name':'Reuven', 'shoesize':46}
----> 3 myfunc(100, info)

TypeError: myfunc() takes 1 positional argument but 2 were given
```

[252]:
```
# turn our dict into keyword arguments with **
myfunc(100, **info)
```

```
x=100, kwargs={'name': 'Reuven', 'shoesize': 46}
```

13 Exercise: XML

Write a function, `xml`: - At a minimum, we need to pass one argument, a string, the opening/closing tag - Optionally, we can pass text (a string) that will be placed inside of the tag - Optionally, we can pass keyword arguments that will be the attributes in the opening tag

Example:

```
xml('tagname')   # returns '<tagname></tagname>'
xml('tagname', 'hello')   # returns '<tagname>hello</tagname>'
xml('tagname', 'hello', x=1, y=2)  # returns <tagname x="1" y="2">hello</tagname>
xml('a', xml('b', 'two tags!'))   # returns <a><b>two tags!</b></a>
```

[261]:
```
def xml(tagname, text='', **kwargs):
    attributes = ''
    for key, value in kwargs.items():
        attributes += f' {key}="{value}"'
    return f'<{tagname}{attributes}>{text}</{tagname}>'
```

[262]:
```
xml('name')   # returns '<tagname></tagname>'
```

[262]:
```
'<name></name>'
```

[263]:
```
xml('name', 'Reuven')
```

[263]:
```
'<name>Reuven</name>'
```

[264]:
```
xml('a',
    xml('b', 'two tags!'))
```

[264]:
```
'<a><b>two tags!</b></a>'
```

[265]:
```
xml('tagname', 'hello', x=1, y=2)
```

[265]:
```
'<tagname x="1" y="2">hello</tagname>'
```

[ ]:
```

```

14 Parameter types

1. Mandatory parameters – either positional or keyword
2. Optional parameters (with default) – either positional or keyword
3. `*args`, with all unclaimed positional arguments
4. `**kwargs`, with all unclaimed keyword arguments

[268]:
```
def myfunc(a, b=100, *args):
    return f'{a=}, {b=}, {args=}'
```

[269]:
```
myfunc(10, 20, 30, 40, 50)
```

[269]:
```
'a=10, b=20, args=(30, 40, 50)'
```

[270]:
```
# allow b to keep its default, but assign values to a and args

# I can use a keyword-only parameter
def myfunc(a, *args, b=100):
    return f'{a=}, {b=}, {args=}'

myfunc(10, 20, 30, 40, 50)
```

[270]:
```
'a=10, b=100, args=(20, 30, 40, 50)'
```

[271]:
```
myfunc(10, 20, 30, 40, 50, b=999)
```

[271]:
```
'a=10, b=999, args=(20, 30, 40, 50)'
```

[272]:
```
# that was a keyword-only parameter with a default
# we can also set keyword-only parameters without defaults

def myfunc(a, *args, b):   # b no longer has a default
    return f'{a=}, {b=}, {args=}'

myfunc(10, 20, 30, 40, 50)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[272], line 7
      4 def myfunc(a, *args, b):   # b no longer has a default
      5     return f'{a=}, {b=}, {args=}'
----> 7 myfunc(10, 20, 30, 40, 50)

TypeError: myfunc() missing 1 required keyword-only argument: 'b'
```

[ ]:
```

```

15 Parameter types

1. Mandatory parameters – either positional or keyword
2. Optional parameters (with default) – either positional or keyword
3. `*args`, with all unclaimed positional arguments
4. `**kwargs`, with all unclaimed keyword arguments
5. Mandatory keyword-only parameter
6. Optional keyword-only parameter (with default)

[273]:
```
myfunc.__code__.co_kwonlyargcount
```

[273]: 1

[274]:
```
myfunc.__code__.co_varnames
```

[274]: ('a', 'b', 'args')

[275]:
```
myfunc.__code__.co_argcount
```

[275]: 1

[284]:
```
s = 'abc'
mylist = [10, 20, 30]

def myfunc(**kwargs):
    for key, value in kwargs.items():
        print(f'{key}: {value}')

myfunc(**dict(zip(s, mylist)))
```

```
a: 10
b: 20
c: 30
```

[287]:
```
# zip takes according to the shortest!
s = 'abc'
mylist = [10, 20, 30, 40, 50]

list(zip(s, mylist))
```

[287]: [('a', 10), ('b', 20), ('c', 30)]

[289]:
```
list(zip(s, mylist, strict=True))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[289], line 1
----> 1 list(zip(s, mylist, strict=True))

ValueError: zip() argument 2 is longer than argument 1
```

[ ]:

16 Parameter types

1. Positional-only arguments, before a `/`
2. Mandatory parameters – either positional or keyword
3. Optional parameters (with default) – either positional or keyword

4. `*args`, with all unclaimed positional arguments (or `*`, if a separator is needed)
5. `**kwargs`, with all unclaimed keyword arguments
6. Mandatory keyword-only parameter
7. Optional keyword-only parameter (with default)

[290]:
```
len('abcd')
```

[290]:
```
4
```

[291]:
```
help(len)
```

```
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

[292]:
```
len(obj='abcd')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[292], line 1
----> 1 len(obj='abcd')

TypeError: len() takes no keyword arguments
```

[295]:
```
def myfunc(name, **kwargs):
    print(f'Hello, {name}. Here are your kwargs:')
    for key, value in kwargs.items():
        print(f'\t{key}:{value}')
```

[296]:
```
myfunc('Reuven', a=100, b=200, c=300)
```

```
Hello, Reuven. Here are your kwargs:
        a:100
        b:200
        c:300
```

[297]:
```
myfunc('Reuven', a=100, b=200, c=300, name='xyz')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[297], line 1
----> 1 myfunc('Reuven', a=100, b=200, c=300, name='xyz')

TypeError: myfunc() got multiple values for argument 'name'
```

[298]:
```
# We use a / to indicate that name is positional only

def myfunc(name, /, **kwargs):
    print(f'Hello, {name}. Here are your kwargs:')
    for key, value in kwargs.items():
        print(f'\t{key}:{value}')
```

[299]:
```
myfunc('Reuven', a=100, b=200, c=300, name='xyz')
```

```
Hello, Reuven. Here are your kwargs:
        a:100
        b:200
        c:300
        name:xyz
```

[302]:
```
# how can I make kw a keyword-only parameter?
# I can use * by itself to indicate that following parameters are keyword only
def myfunc(reg, *, kw):
    print(f'{reg=}, {kw=}')
```

[303]:
```
myfunc(10, 20)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[303], line 1
----> 1 myfunc(10, 20)

TypeError: myfunc() takes 1 positional argument but 2 were given
```

[304]:
```
myfunc(10, kw=20)
```

```
reg=10, kw=20
```

[305]:
```
help(str.split)
```

```
Help on method_descriptor:

split(self, /, sep=None, maxsplit=-1)
    Return a list of the substrings in the string, using sep as the separator
string.

      sep
        The separator used to split the string.

        When set to None (the default value), will split on any whitespace
        character (including \n \r \t \f and spaces) and will discard
        empty strings from the result.
      maxsplit
```

```
        Maximum number of splits (starting from the left).
        -1 (the default value) means no limit.

    Note, str.split() is mainly useful for data that has been intentionally
    delimited.  With natural text that includes punctuation, consider using
    the regular expression module.
```

[306]:
```
x,*y,z = mylist = [10, 20, 30, 40, 50, 60]
```

[310]:
```
def hello(name:str) -> str:   # type hint / type annotation
    return f'Hello, {name}!'
```

[311]:
```
hello.__annotations__
```

[311]:
```
{'name': str, 'return': str}
```

[312]:
```
hello(5)
```

[312]:
```
'Hello, 5!'
```

[ ]:
```

```