List, Tuples and Dictionaries

Sebastian Proksch

University of Zurich, Department of Informatics

Date of the Midterm

- (See courses.uzh.ch for the definitive answer ;))

- November 4, 2019
- Start: ~6:15pm
- Duration:
  - 60min (Info1 Students)
  - 90min (DOP Students)

- Details will be announced in the week before the exam.

Outline for Today

- Python Structured Data Types
  - Tuples
  - Lists
  - Dictionaries
- Example exercises

Which Data Types Have We Seen So Far?

int, float, boolean, string, (function, type)

These are mostly *primitive* types, basic building blocks that allow fundamental operations.

Group Related Operations?

```
def sum_division(a, b, c):
    x = a + b
    return x // c

def sum_modulo(a, b, c):
    x = a + b
    return x % c

div = sum_division(1, 4, 2)
mod = sum_modulo(1, 4, 2)
```

A function can only return a single value!

We Need More Complex Data Types

To write useful programs we need advanced types that provide us with the flexibility to compose those basic types into more complex data structures.

Tuples

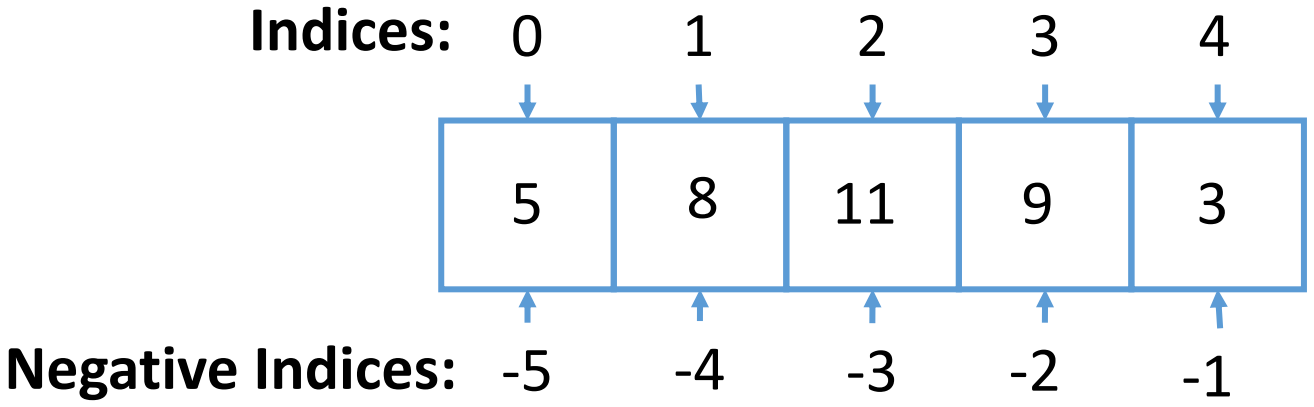
Tuples Are An Ordered Set Of Values

Each value can be accessed by an index or through ranges.

```
t = (5, 8, 11, 9, 3)

x = t[1] # 8
```

```
a = t[0:2]      # (5, 8)
b = t[:2]       # (5, 8)

c = t[1:2]      # (8)
d = t[1]        # 8

e = t[2:4]      # (11, 9)
f = t[-2:-1]    # (9)
```

How To Access Elements Of A Tuple




```
t = (5, 8, 11, 9, 3)
print(t[0])             # prints the first element
print(t[-1])            # prints the last element
print(t[len(t)])        # IndexError
print(t[len(t)-1])      # prints the last element
print(t[2])             # prints 11
```

Tuple Values Can Be Of Any Type

A single tuple can even contain mixed values.

```
t = (1, 2.34, ‘foo’, print)

print(t[1]) # 2.34
```

Tuples Can Be Nested

Elements can be values with an arbitrary type. This means that it is possible to also add other tuples (lists, dictionaries, ...) as elements.

```
t = (('one', 1), ('two', 2), ('three', 3))

print(t[0])    # prints ('one', 1)
print(t[0][0]) # prints one
print(t[0][1]) # prints 1

print(t[0][0][0]) # ??
```

Creating Tuples

```
t1 = 1, 2, 3, 4
t2 = (1, 2, 3, 4)
t3 = (1, 2, 'three', 5.8, (1, 1))

# creating single element tuples
t4 = (1,)
t5 = (1)

print(type(t4))    # prints <class 'tuple'>
print(type(t5))    # prints <class 'int'>
```

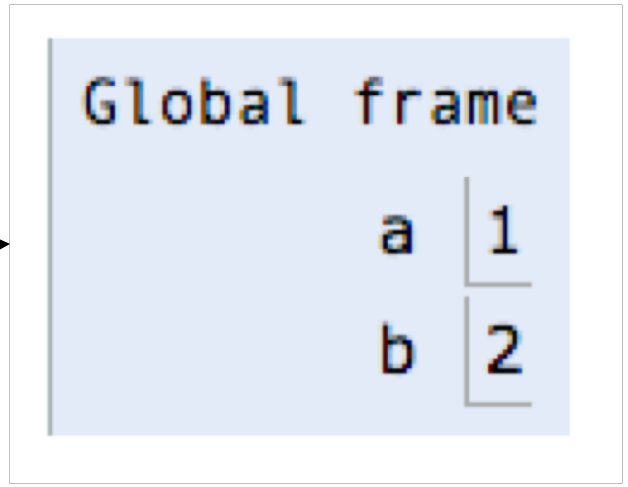
Using A Tuple In An Assignment

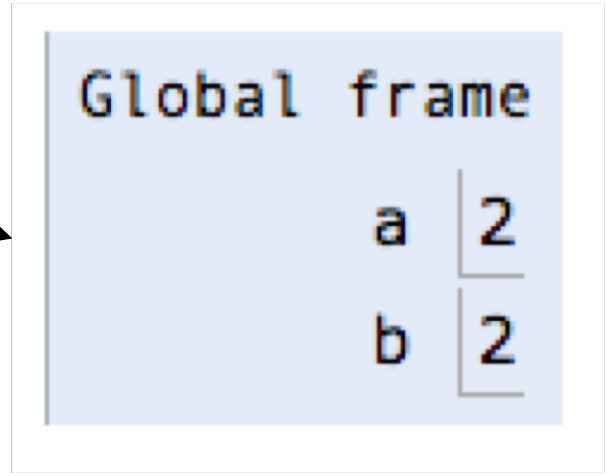
```
(a, b) = (1, 2)

print(a) # 1
print(b) # 2
```

Pecking

Swapping Values

```
# initialize
a = 1
b = 2

# swap?!
a = b
b = a

# result
print(a) # 2
print(b) # 2
```





Swapping Values – For Real

```
a = 1
b = 2

# swap values – the “traditional” approach
temp = a
a = b
b = temp


 (a, b) = (b, a)
```

Iterating over tuples

Tuples are **iterable** and can be used in loops.

```
t = (1, 2, 3, 4)

for item in t:
    print(item) # prints 1 2 3 4
```

Assigning Values To A Tuple Index

- You cannot modify the content of a tuple once you created it, tuples are “immutable”.
- You can create a new tuple with the desired content

```
t1 = (1, 2, 3)

t1[0] = 4 # TypeError: 'tuple' object does not support item assignment

t2 = (4, 5)
t3 = t1 + t2

print(t3) # (1, 2, 3, 4, 5)
```

Returning “multiple” values from functions

A function can return a tuple to **pack** multiple values into one return value. Python can automatically **unpack** a tuple on assign.

```
def first_and_last(numbers):
    # or return numbers[0], numbers[-1]
    # the parentheses are not necessary
    return (numbers[0], numbers[-1])

a = first_and_last((1, 2, 3, 4)) # (1, 4)
b = first_and_last((1,))         # (1, 1)
(f,l) = first_and_last((7, 8))    # f=7, l=8
```

Grouping Related Operations -- Revisited

```
def sum_division(a, b, c):
      x = a + b
      return x // c

def sum_modulo(a, b, c):
      x = a + b
      return x % c

div = sum_division(1,4,2)
mod = sum_modulo(1,4,2)
```

```
def sum_both(a, b, c):
    x = a + b
    div = x // c
    mod = x % c
    return (div, mod)

div, mod = sum_both(1,4,2)
```

Lists

Lists

A list, similarly to a tuple, is an ordered set of values (or elements), where each value is identified by an index. The contained values can be of any type and a single list can even contain mixed values. The main difference between the two data structures is that lists are **mutable**.

```
>>> [1, 2, 3, 4] # notice “[…]” instead of “(…)”
>>> [1, 'hello', 3.14]  # mixed values
>>> [1, 2, [3, 4, [5, 6]]]  # nested lists
>>> [] # empty list
```

How to access the elements of a lists

```
numbers = [6, 3, 1, 7]

a = numbers[1]  # second element of the list
b = numbers[0]  # first element of the list
c = numbers[4]  # IndexError: index out of range
d = numbers[-1] # prints the last element

l1 = l[0:2]   # [6, 3]
l2 = l[2:4]   # [1, 7]
l3 = l[:]     # copies the original list
l4 = l[0:4]   # as before, creates a copy
```

Iterating elements of a list/tuple/string/...

Python can iterate over every **Iterable**. However, additional logic is required to have access to the index of an element in the iterable.

```
l = [2, 7, 5, 'two', 'seven', 11]

for el in l: #1
    print(el)

for index in range(len(l)): #2
    print("lst[{}] = {}".format(index, l[index]))

for index, val in enumerate(l): #3
     print("lst[{}] = {}").format(index, val)
```

Lists Are Mutable

```
l = [2, 7, 5, 'two', 'seven', 11]
print(l) # prints [2, 7, 5, 'two', 'seven', 11]

l[2] = 0 # NO ERROR!!!
print(l) # prints [2, 7, 0, 'two', 'seven', 11]

# here we are creating a new list object and assign
# it to the variable l, we DO NOT modify the content
# of the original list!!!
l = [1, 2, 3]
```

Useful Operations On Lists

- lists are Python objects that contain values and functions (more about it later)
- to access the values of a list, you can use the [] operator
- to access the methods(functions) of a list, you can use the . operator

```
l = [2, 5, 1]
print(l) # prints [2, 5, 1]

# append an element at the end of the list object
l.append(3)

print(l) # prints [2, 5, 1, 3]
```

Useful Operations On Lists (2)

```
l = [2, 7, 5, 'two', 'seven', 11]

# list membership
print(2 in l) # True
print(10 in l) # False
print(10 not in l) # True

# concatenation
l2 = [7, 8, 19]
l3 = l + l2
print(l3) # [2, 7, 5, 'two', 'seven', 11, 7, 8, 9]

# length
print(len(l3)) # 9

# repetition
print(3 * [2, 3]) # [2, 3, 2, 3, 2, 3]
```

Useful Operations On Lists (3)

```
l = [“a”, ”b”, “c”]

# concatenating two list
l.extend([“d”, “e”]) # l: [a, b, c, d, e]

# removals
e = l.pop()   # return: c -- l: [a, b, c, d])
del(l[0])     # return: None -- l: [b, c, d])
l.remove(“c”) # return: None -- l: [b, d]
```

Other operations on lists

```
lst2 = [3, 7, 1, -1]

lst2.sort() # modifies the list in place!!!
print(lst2) # [-1, 1, 3, 7]
lst2.reverse()
print(lst2) # [7, 3, 1, -1]

lst3 = [5, -1, 10, 2]
lst4 = sorted(lst3) # creates another list and sorts it!!!
print(lst3) # [5, -1, 10, 2]
print(lst4) # [-1, 2, 5, 10]
```

Equality, Aliasing, Clones

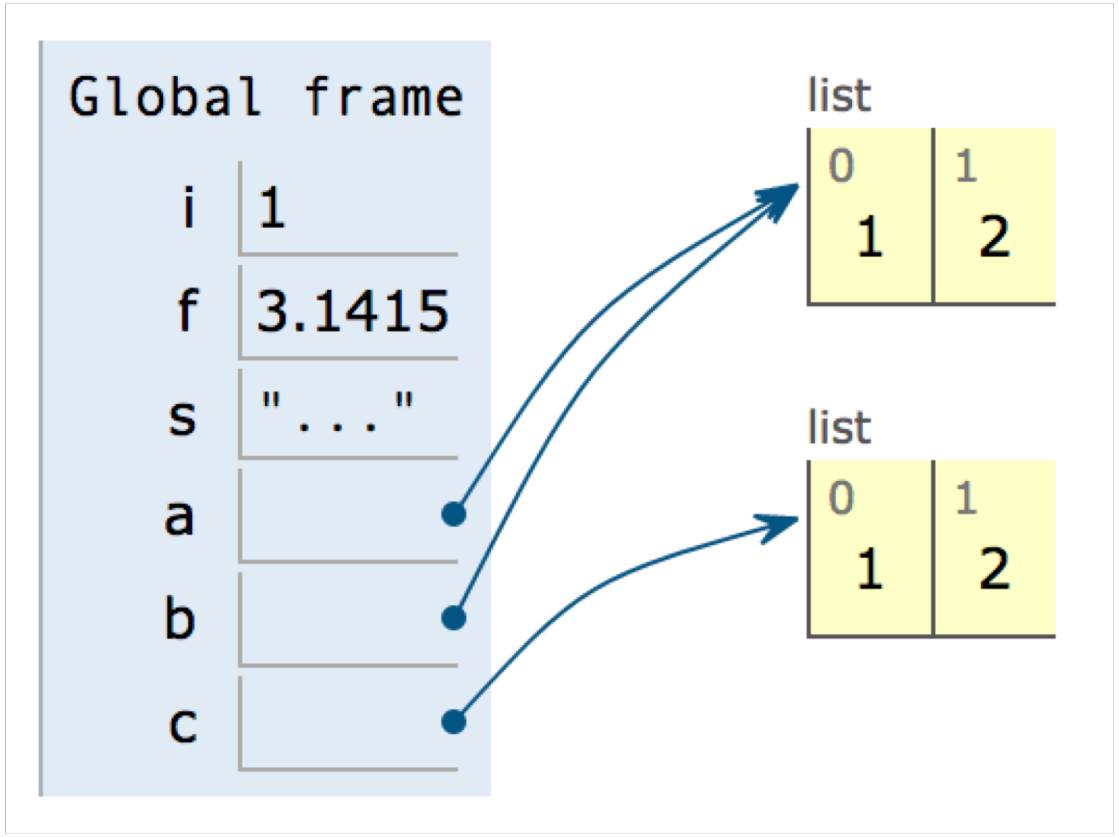
Equality And Identity Operators

== checks, if two objects are *equal*

is checks, if two objects are *the same*

Equality Examples

```
i = 1
f = 3.1415
s = "…"

a = [1, 2]
b = a    # alias
c = a[:] # clone

a == b # True
a == c # True

a is b # True
a is c # False
```



http://www.pythontutor.com

Aliasing

```
a = [1, 2, 3]
b = a # b is now an alias for a

print(a) # [1, 2, 3]
print(b) # [1, 2, 3]

a[0] = 10
print(a) # [10, 2, 3]

print(b) # the same as for a, both point to the same object!
```

Using A Range Selector Clones A List

```
a = [2, 5, 1]
b = a[:] # equal to a[0:3]

print(a) # [1, 2, 3]
print(b) # [1, 2, 3]

a[0] = 10
print(a) # [10, 2, 3]
print(b) # [1, 2, 3], “a” and “b” point to dif. obj.
```

Cloning is shallow

Try a deep clone

```
a = [[1], [2, 3], [4, 5, 6]]
b = a[:]

print(a) # [[1], [2, 3], [4, 5, 6]]
print(b) # [[1], [2, 3], [4, 5, 6]]

a[0] = “x”
a[1][0] = 10

print(a) # [“x”, [10, 3], [4, 5, 6]]
print(b) # [[1], [10, 3], [4, 5, 6]] – same obj
```

Lists As Function Parameters

- passing a list as an argument, passes a reference to the list, not a copy; if the content of the list is modified in the function, the changes will be reflected in the original list.

```
x = [1, 2, 3]

def foo(l):
    l.append(4)

print(x) # [1, 2, 3]
foo(x)
print(x) # [1, 2, 3, 4]
```

Strings

Strings and Lists – str.split

```
s1 = "a simple string"

chars1 = s1.split() # defau
print(chars1) # ['a', 'simple', 'string']

s2 = "another_simple_string"

chars2 = s2.split("_") # split by "_”
print(chars2) # ['another', 'simple', 'string']
```

Remember, by using the dot (.) notation we are calling a function on the str object.

Strings and Lists – str.join

```
chars = ['a', 'simple', 'string']

# join the elements of the list
str = "".join(chars)
print(str) # “asimplestring”

# join the elements of the list using
# the specified character
str2 = " ".join(chars)
print(str2) # “a simple string”
```

Dictionaries

Problem: Using Related Information

```
names = [‘Alfred’, ‘Becky’, …]
phones = [‘078 123 45 67’, ‘078 910 12 34’, …]

def print_details(name, names, numbers):
  idx = names.index(name)
  print(“%s: %s” % (names[idx], numbers[idx]))

print_details(“Alfred”, names, phones)
```

Dictionaries - Motivation

- Strings, lists and tuples allow you to store sequences of values and than access them based on the position in the sequence (the index)
- Sometimes we would like to store sequences of values and than retrieve a particular value using something else than the index (ex. phone book and looking up the phone number of a particular person)
- This can be achieved with a Python **dictionary**

Dictionaries

- a dictionary is a Python data type that stores pairs of data of the form: key-value
- a value can be retrieved using the corresponding key
- the key-value pairs are **not ordered**

```
phone_book = {} # empty dictionary
phone_book = {‘Alfred’: ‘078…’} # non-empty dict.

# adding an entry
phone_book[‘Becky’] = ‘078 910 12 34’

# retrieving Alfred's phone number
print(phone_book[‘Alfred’])

# overriding an entry
phone_book[‘Alfred’] = ‘123’
```

Comparison Of Lists And Dictionaries

List

Index	Value
0	'Anna'
1	'Andre'
2	'Alina'
4	'Adrian'

Dictionary

Key	Value
'Anna'	'072 182 398'
'Andre'	'077 293 100'
'Alina'	'056 234 100'
'Adrian'	'078 122 111'

Properties of Dictionary Keys and Values

- **Keys**:
  - Unique
  - Immutable type (int, float, string, tuple, bool)
- **Values**:
  - Can be duplicates
  - Any value type (even other dictionaries or lists)

Operations On Dictionaries (1)

```
# continuation from the previous slide

# test if a name is in the phone book
if ’Alfred' in phone_book: # prints “Found!”
    print('Found!')
else:
    print('Not found!')

# deleting an entry
del(phone_book['Alfred'])

if 'Alfred' in phone_book: # prints “Not found!”
    print('Found!')
else:
    print('Not found!')
```

Operations On Dictionaries (2)

```
def print_details(name, num):
  print('number of %s is %s' % (name, num))

# implicit iteration over phonebook elements
for name in phone_book:
  print_details(name, phone_book[name]))

# iterate over the keys explicitly:
for name in phone_book.keys():
  print_details(name, phone_book[name]))

# iterating over the values
for num in phone_book.values():
  print_details("??", num))
```

Hint: Read Official Documentation

https://docs.python.org/3/tutorial/datastructures.html

5. Data Structures ¶

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

`list.`**`append`**(*x*)
Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

`list.`**`extend`**(*iterable*)
Extend the list by appending all the items from the iterable. Equivalent to `a[len(a):] = iterable`.

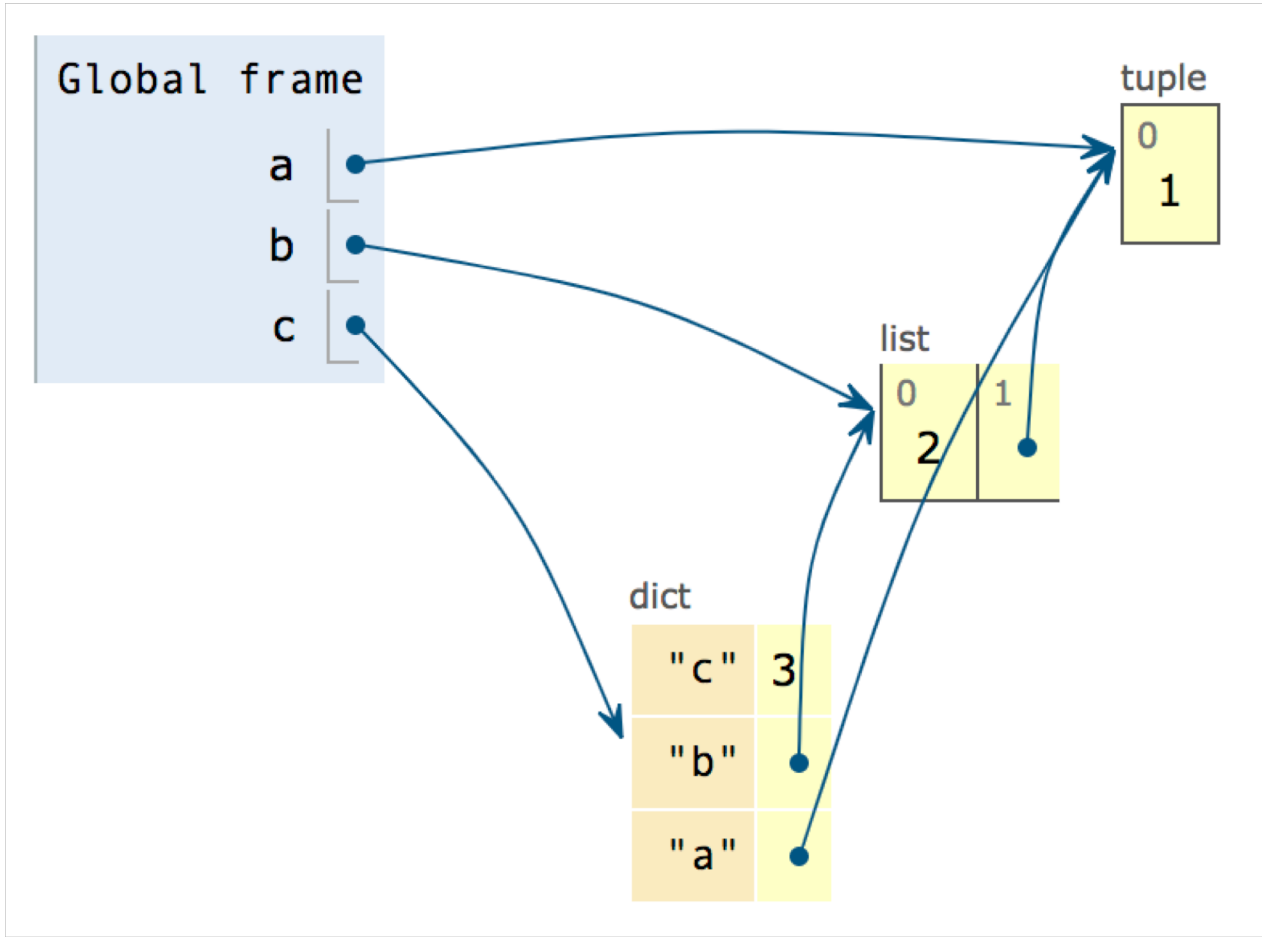
`list.`**`insert`**(*i*, *x*)
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.`**`remove`**(*x*)
Remove the first item from the list whose value is *x*. It is an error if there is no such item.

`list.`**`pop`**([*i*])

Tuple, List, and Dictionary are Reference Types

```
a = (1,)
b = [2]
c = { “c”: 3 }

b.append(a)
c[“b”] = b
c[“a”] = a
```



Take Home Message

After todays lecture, you should know...

- The built-in data structures Tuple, List, Dictionary
- Accessing elements of these data structures
- How to iterate these data structures
- How to use range selectors
- The difference between a Tuple and a List
- The difference between a value type and a reference type
- The difference between mutability and immutability
- Several helpful operations on List, Tuple, Dictionary
- Basic understanding of aliasing and equality

Exercise 1

Find Minimum

Exercise 1

Implement a function that receives a list of numbers as an argument and returns the position of the minimum value and the actual minimum value. Return None for an empty list.

Example:

find_min([3, 2, 5, 1, 7]) → (3, 1)

Exercise 1: Sample Solution

```
def find_min(numbers):
    # True if numbers == None or numbers == []
    if not numbers:
        return None
    min_value = numbers[0]
    min_index = 0
    for index in range(1, len(numbers)):
        if numbers[index] < min_value:
            min_value = numbers[index]
            min_index = index
    return min_index, min_value

print(find_min([])) # None
print(find_min([1])) # (0, 1)
print(find_min([2, 3, -1, 5, 2])) # (2, -1)
print(find_min([-1, 2, 3, 4])) # (0, -1)
print(find_min([10, 2, 4, -1])) # (3, -1)
```

Exercise 2

Matrix

Exercise 2

Implement a function that receives a matrix (list of lists) of numbers as an argument and returns all non-zero entries. The return value should be dictionary which has the indices (row & column) of the number as keys (use a tuple) associated with the non-zero values.

Example:

[[0, 0, 1],
  [2, 0, 0],
  [0, 5, 0]]

–> { (0, 2): 1, (1, 0): 2, (2, 1): 5 }

Exercise 2: Sample Solution

```
def build_sparse_matrix(matrix):
    sparse_matrix = {}
    # we assume the matrix is built correctly,
    # each row has the same length
    for row in range(len(matrix)):
        for col in range(len(matrix[row])):
            if matrix[row][col] != 0:
                key = (row, col)
                value = matrix[row][col]
                sparse_matrix[key] = value
    return sparse_matrix

print(build_sparse_matrix(
    [[0, 0, 1], [2, 0, 0], [0, 5, 0]]))
# {(1, 0): 2, (0, 2): 1, (2, 1): 5}
# the exact order of the key-value pairs may differ
```