Data Visualization

BTech Computer Science Stream, January 2025

Week 2 Python Language Basics - Demonstration Code

Name: Manoj R, Reg Number, Date: 24/12/2024, Modified by: Shwetha Rai

This Notebook demonstrates Python's workhorse data structures: tuples, lists, dicts, and sets and discuss creating your own reusable Python functions

Following naming conventions are used for Python's data structures

tuple- tup
Sequence- seq
list- variablename_list
dicts- dict
sets- variablename_set

Tuple

A Python Tuple is immutable i.e., once assigned, it cannot be changed. Applications can be found in storing the coordinates in multidimensional space

```
tup = 4, 5, 6 # without parenthesis considered as tuple
tup
```

```
(4, 5, 6)
```

```
tup = (4, 5, 6) #parenthesis for tuple
tup
```

```
(4, 5, 6)
```

Defining tuples in more complicated expressions, it's often necessary to enclose the values in parentheses, as in this example of creating a tuple of tuples

```
tup = tuple([4, 0, 2])
tup
```

```
(4, 0, 2)
```

```
tup = tuple('string')
tup
```

```
('s', 't', 'r', 'i', 'n', 'g')
```

**Elements can be accessed with square brackets [ ] as with most other sequence types. As in C, C++, Java, and many other languages,
sequences are 0-indexed in Python**

```
tup[4]
```

```
'n'
```

```
type(tup[0]) # Check the type of the value in the tuple
```

```
str
```

Nested tuples

```
nested_tup = (4, 5, 6), (7, 8)
print(nested_tup)
print(nested_tup[0])
print(nested_tup[1])
```

```
((4, 5, 6), (7, 8))
(4, 5, 6)
(7, 8)
```

The objects stored in a tuple may be mutable themselves, once the tuple is created it's not possible to modify which object is stored in each slot

```
tup = tuple(['foo', [1, 2], True])
type(tup[2])
#tup[2] = False
```

```
bool
```

If an object inside a tuple is mutable, such as a list, you can modify it in place

```
tup[1].append(3)
tup
```

```
('foo', [1, 2, 3], True)
```

You can concatenate tuples using the + operator to produce longer tuples

```
(4, None, 'foo') + (6, 0) + ('bar',)
```

```
(4, None, 'foo', 6, 0, 'bar')
```

Multiplying a tuple by an integer, as with lists, has the effect of concatenating together that many copies of the tuple

```
('foo', 'bar')*4
```

```
('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

Unpacking tuples

If you try to assign to a tuple-like expression of variables, Python will attempt to unpack the value on the righthand side of the equals sign

```
tup = (4, 5, 6)
a, b, c = tup
b
```

```
5
```

Sequences with nested tuples can be unpacked

```
tup = 4, 5, (6, 7)
a, b, (c, d) = tup
d
```

```
7
```

Using this functionality you can easily swap variable names, a task which in many languages might look like:

```
tmp = a
a = b
b = tmp
```

But, in Python, the swap can be done like this:

```
a, b = 1, 2
a
b
b, a = a, b
a
b
```

```
1
```

Formatted string literals: f or F strings

```
a = 1
b = 2
c = 3
print(f'a={a}, b={b}, c={c}')
```

```
a=1, b=2, c=3
```

A common use of variable unpacking is iterating over sequences of tuples or lists

```
seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
for a, b, c in seq:
    print(f'a={a}, b={b}, c={c}')
#print(f'a={a}, b={b}, c={c}')
```

```
a=1, b=2, c=3
a=4, b=5, c=6
a=7, b=8, c=9
```

More advanced tuple unpacking to help with situations where you may want to "pluck" a few elements from the beginning of a tuple. This uses the special syntax **rest*, which is also used in function signatures to capture an arbitrarily long list of positional arguments

```
values = 1, 2, 3, 4, 5
a, b, *rest, c = values
print(a)
print(b)
print(rest)
print(c)
```

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

This rest bit is sometimes something you want to discard; there is nothing special about the rest name. **As a matter of convention, many Python programmers will use the underscore (_) for unwanted variables:**

```
a, b, *_ = values
print(_)
```

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

Tuple methods

Since the size and contents of a tuple cannot be modified, it is very light on instance methods. min, max, sum, sorted. A particularly useful one (also available on lists) is count, which counts the number of occurrences of a value:

```
a = (1, 2, 2, 2, 3, 4, 2)
a.count(2)
len(a)
```

```
7
```

Deleting Tuples

```
del a
#len(a)
```

List

In contrast with tuples, lists are variable-length and their contents can be modified in-place. You can define them using square brackets [ ] or using the list type function:

```
a_list = [2, 3, 7, None]
print(a_list)
tup = ("foo", "bar", "baz")
b_list = list(tup)
b_list
b_list[1] = "peekaboo"
b_list
```

```
[2, 3, 7, None]
```

```
['foo', 'peekaboo', 'baz']
```

Lists and tuples are semantically similar (though tuples cannot be modified) and can be used interchangeably in many functions. The list function is frequently used in data processing as a way to materialize an iterator or generator expression:

```
gen = range(10)
gen
list(gen)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Adding and removing elements:
Elements can be appended to the end of the list with the append method:

```
b_list.append("dwarf")
b_list
```

```
['foo', 'peekaboo', 'baz', 'dwarf']
```

Insert is computationally expensive compared with append, because references to subsequent elements have to be shifted internally to make room for the new element.
If you need to insert elements at both the beginning and end of a sequence, you may wish to explore collections. deque, a double-ended queue, for this purpose.
Using insert you can insert an element at a specific location in the list: The insertion index must be between 0 and the length of the list, inclusive

```
b_list.insert(1, "red")
b_list
```

```
['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

The inverse operation to insert is pop, which removes and returns an element at a particular index

```
b_list.pop(2)
b_list
```

```
['foo', 'red', 'baz', 'dwarf']
```

If performance is not a concern, by using append and remove, you can use a Python list as a set-like data structure

```
#b_list.append("foo")
b_list
b_list.remove("foo")
b_list
```

```
['red', 'baz', 'dwarf']
```

Check if a list contains a value using the in keyword:

Checking whether a list contains a value is a lot slower than doing so with dicts and sets , as Python makes a linear scan across the values of the list, whereas it can check the others (based on hash tables) in constant time.

```
"dwarf" in b_list
```

```
True
```

The keyword not can be used to negate in

```
"dwarf" not in b_list
```

```
False
```

Concatenating and combining lists

Similar to tuples, adding two lists together with + concatenates them:

```
[4, None, "foo"] + [7, 8, (2, 3)]
```

```
[4, None, 'foo', 7, 8, (2, 3)]
```

If you have a list already defined, you can append multiple elements to it using the extend method

```
x = [4, None, "foo"]
x.extend([7, 8, (2, 3)])
x
```

```
[4, None, 'foo', 7, 8, (2, 3)]
```

Sorting

You can sort a list in-place (without creating a new object) by calling its sort function:

```
a = [7, 2, 5, 1, 3]
a.sort()
a
```

```
[1, 2, 3, 5, 7]
```

Sort has a few options that will occasionally come in handy. One is the ability to pass a secondary sort key i.e., a function that produces a value to use to sort the objects. For example, we could sort a collection of strings by their lengths:

```
b = ["saw", "small", "He", "foxes", "six"]
b.sort(key=len)
b
```

```
['He', 'saw', 'six', 'small', 'foxes']
```

Slicing

You can select sections of most sequence types by using slice notation, which in its basic form consists of *start:stop* passed to the indexing operator [ ]

```
seq = [7, 2, 3, 7, 5, 6, 0, 1]
seq[1:5]
```

```
[2, 3, 7, 5]
```

Slices can also be assigned to replace values in a sequence

```
seq[3:5] = [6, 3]
seq
```

```
[7, 2, 3, 6, 3, 6, 0, 1]
```

While the element at the start index is included, the stop index is not included, so that the number of elements in the result is stop - start. Either the start or stop can be omitted, in which case they default to the start of the sequence and the end of the sequence, respectively

```
seq[:5]
seq[3:]
```

```
[6, 3, 6, 0, 1]
```

Negative indices slice the sequence relative to the end

```
seq[-4:]
seq[-6:-2]
```

```
[3, 6, 3, 6]
```

Slicing semantics takes a bit of getting used to, especially if you're coming from R or MATLAB. the indices are shown at the "bin edges" to help show where the slice selections start and stop using positive or negative indices. A step can also be used after a second colon to, say, take every other element:

```
seq[::2]
```

```
[7, 3, 3, 0]
```

```
seq[::-1]
```

```
[1, 0, 6, 3, 6, 3, 2, 7]
```

dict

dict may be the most important built-in Python data structure. In other programming languages, dicts are sometimes called hash maps or associative arrays.

- A dict is an unordered collection of key-value pairs,where key and value are Python objects.
- Each key is associated with a value so that a value can be conveniently retrieved, inserted, modified, or deleted given a particular key.
- One approach for creating one is to use curly braces {} and colons to separate keys and values:

```
empty_dict = {}
d1 = {"a": "some value", "b": [1, 2, 3, 4]}
d1
```

```
{'a': 'some value', 'b': [1, 2, 3, 4]}
```

You can access, insert, or set elements using the same syntax as for accessing elements of a list or tuple

```
d1[7] = "an integer"
d1
d1["b"]
```

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

You can check if a dict contains a key using the same syntax used for checking whether a list or tuple contains a value

```
"b" in d1
```

```
True
```

You can delete values either using the del keyword or the pop method (which simultaneously returns the value and deletes the key):

```
d1[5] = "some value"
d1
d1["dummy"] = "another value"
d1
del d1[5]
d1
ret = d1.pop("dummy")
ret
d1
```

```
{'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

The keys and values method give you iterators of the dict's keys and values, respectively. The order of the keys depends on the order of their insertion, and these functions output the keys and values in the same respective order

```
list(d1.keys())
list(d1.values())
```

```
['some value', [1, 2, 3, 4], 'an integer']
```

```
list(d1.items())
```

```
[('a', 'some value'), ('b', [1, 2, 3, 4]), (7, 'an integer')]
```

You can merge one dict into another using the update method

```
d1.update({"b": "foo", "c": 12})
d1
```

```
{'a': 'some value', 'b': 'foo', 7: 'an integer', 'c': 12}
```

Creating dicts from sequences It's common to occasionally end up with two sequences that you want to pair up element-wise in a dict. As a first cut, you might write code like this:

```
tuples = zip(range(5), reversed(range(5)))
tuples
mapping = dict(tuples)
mapping
```

```
{0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

Default values
It's common to have logic like:

```
if key in some_dict:
    value = some_dict[key]
else:
    value = default_value
```

Thus, the dict methods get and pop can take a default value to be returned, so that the above if-else block can be written simply as:

```
value = some_dict.get(key, default_value)
```

get by default will return None if the key is not present, while pop will raise an exception. With setting values, it may be that the values in a dict are another kind of collection, like a list. For example, you could imagine categorizing a list of words by their first letters as a dict of lists:

```
words = ["apple", "bat", "bar", "atom", "book"]
by_letter = {}

for word in words:
    letter = word[0]
    if letter not in by_letter:
        by_letter[letter] = [word]
    else:
        by_letter[letter].append(word)

by_letter
```

```
{'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

The setdefault dict method can be used to simplify this workflow. The preceding for loop can be rewritten as:

```
for word in words:
    letter = word[0]
    by_letter.setdefault(letter, []).append(word)
```

The built-in collections module has a useful class, defaultdict, which makes this even easier. To create one, you pass a type or function for generating the default value for each slot in the dict

```
by_letter = {}
for word in words:
    letter = word[0]
```

```
    by_letter.setdefault(letter, []).append(word)
by_letter
```

```
{'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

```
from collections import defaultdict
by_letter = defaultdict(list)
for word in words:
    by_letter[word[0]].append(word)
```

Valid dict key types

While the values of a dict can be any Python object, the keys generally have to be immutable objects like scalar types (int, float, string) or tuples (all the objects in the tuple need to be immutable, too).
The technical term here is hashability.
You can check whether an object is hashable (can be used as a key in a dict) with the hash function

```
hash("string")
hash((1, 2, (2, 3)))
#hash((1, 2, [2, 3])) # fails because lists are mutable
```

```
-9209053662355515447
```

To use a list as a key, one option is to convert it to a tuple, which can be hashed as long as its elements also can

```
d = {}
d[tuple([1, 2, 3])] = 5
d
```

```
{(1, 2, 3): 5}
```

##set

A set is an unordered collection of unique elements. You can think of them like dict keys, but keys only, no values.
A set can be created in two ways:

- via the set function or
- via a set literal with curly braces

```
set([2, 2, 2, 1, 3, 3])
{2, 2, 2, 1, 3, 3}
```

```
{1, 2, 3}
```

```
a = {1, 2, 3, 4, 5}
b = {3, 4, 5, 6, 7, 8}
```

The union of these two sets is the set of distinct elements occurring in either set. This can be computed with either the union method or the | binary operator

```
a.union(b)
a | b
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

The intersection contains the elements occurring in both sets. The & operator or the intersection method can be used

```
a.intersection(b)
a & b
```

```
{3, 4, 5}
```

All of the logical set operations have in-place counterparts, which enable you to replace the contents of the set on the left side of the operation with the result. For very large sets, this may be more efficient:

```
c = a.copy()
c |= b
c
d = a.copy()
d &= b
d
```

```
{3, 4, 5}
```

Like a dict's keys, a set's elements generally must be immutable, and they must be hashable (which means that calling hash on a value does not raise an exception). In order to store list-like elements (or other mutable sequences) in a set, you can convert them to tuples

```
my_data = [1, 2, 3, 4]
my_set = {tuple(my_data)}
my_set
```

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

You can also check if a set is a subset of (is contained in) or a superset of (contains all elements of) another set

```
a_set = {1, 2, 3, 4, 5}
{1, 2, 3}.issubset(a_set)
a_set.issuperset({1, 2, 3})
```

```
True
```

Sets are equal if and only if their contents are equal

```
{1, 2, 3} == {3, 2, 1}
```

```
True
```

```
sorted([7, 1, 2, 6, 0, 3, 2])
sorted("horse race")
```

```
[' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

```
seq1 = ["foo", "bar", "baz"]
seq2 = ["one", "two", "three"]
zipped = zip(seq1, seq2)
list(zipped)
```

```
[('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

```
seq3 = [False, True]
list(zip(seq1, seq2, seq3))
```

```
[('foo', 'one', False), ('bar', 'two', True)]
```

```
for index, (a, b) in enumerate(zip(seq1, seq2)):
    print(f"{index}: {a}, {b}")
```

```
0: foo, one
1: bar, two
2: baz, three
```

```
list(reversed(range(10)))
```

```
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

```
strings = ["a", "as", "bat", "car", "dove", "python"]
[x.upper() for x in strings if len(x) > 2]
```

```
['BAT', 'CAR', 'DOVE', 'PYTHON']
```

```
unique_lengths = {len(x) for x in strings}
unique_lengths
```

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

```
set(map(len, strings))
```

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

```
loc_mapping = {value: index for index, value in enumerate(strings)}
loc_mapping
```

```
{'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

```
all_data = [["John", "Emily", "Michael", "Mary", "Steven"],
            ["Maria", "Juan", "Javier", "Natalia", "Pilar"]]
```

```
names_of_interest = []
for names in all_data:
    enough_as = [name for name in names if name.count("a") >= 2]
    names_of_interest.extend(enough_as)
names_of_interest
```

```
['Maria', 'Natalia']
```

```
result = [name for names in all_data for name in names
          if name.count("a") >= 2]
result
```

```
['Maria', 'Natalia']
```

```
some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
flattened = [x for tup in some_tuples for x in tup]
flattened
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
flattened = []

for tup in some_tuples:
    for x in tup:
        flattened.append(x)
```

```
[[x for x in tup] for tup in some_tuples]
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Functions

Functions are the primary and most important method of code organization and reuse in Python. As a rule of thumb, if you anticipate needing to repeat the same or very similar code more than once, it may be worth writing a reusable function. Functions can also help make your code more readable by giving a name to a group of Python statements.

Functions are declared with the def keyword. A function contains a block of code with an optional use of the with the return keyword

```
def my_function(x, y):
    return x + y

my_function(1, 2)
result = my_function(1, 2)
result
```

```
3
```

There is no issue with having multiple return statements. If Python reaches the end of a function without encountering a return statement, None is returned automatically. For example

```
def function_without_return(x):
    print(x)

result = function_without_return("hello!")
print(result)
```

```
hello!
None
```

```
def my_function2(x, y, z=1.5):
    if z > 1:
        return z * (x + y)
    else:
        return z / (x + y)
```

Each function can have positional arguments and keyword arguments. Keyword arguments are most commonly used to specify default values or optional arguments. In the preceding function, x and y are positional arguments while z is a keyword argument. This means that the function can be called in any of these ways:

```
my_function2(5, 6, z=0.7)
my_function2(3.14, 7, 3.5)
my_function2(10, 20)
```

```
45.0
```

Namespaces, Scope, and Local Functions

Functions can access variables created inside the function as well as those outside the function in higher (or even global) scopes.

- An alternative and more descriptive name describing a variable scope in Python is a namespace.
- Any variables that are assigned within a function by default are assigned to the local namespace.
- The local namespace is created when the function is called and immediately populated by the function's arguments.
- After the function is finished, the local namespace is destroyed (with some exceptions that are outside the purview of this chapter).
- Consider the following function: When func() is called,
  - the empty list a is created,
  - five elements are appended, and
  - then a is destroyed when the function exits.
- Suppose instead we had declared a as follows:

```
a = []
def func():
    for i in range(5):
        a.append(i)
```

Each call to func will modify the list a

```
func()
a
func()
a
```

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

Assigning variables outside of the function's scope is possible, but those variables must be declared explicitly either using the global or nonlocal keywords.
nonlocal allows a function to modify variables defined in a higher level scope that is not global.

```
a = None
def bind_a_variable():
    global a
    a = []
bind_a_variable()
print(a)
```

```
[]
```

Functions Are Objects

Since Python functions are objects, many constructs can be easily expressed that are difficult to do in other languages. Suppose we were doing some data cleaning and needed to apply a bunch of transformations to the following list of strings:

```
states = ["   Alabama ", "Georgia!", "Georgia", "georgia", "FlOrIda",
          "south   carolina##", "West virginia?"]

import re

def clean_strings(strings):
    result = []
    for value in strings:
        value = value.strip()
        value = re.sub("[!#?]", "", value)
        value = value.title()
        result.append(value)
    return result

clean_strings(states)
```

```
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South   Carolina',
 'West Virginia']
```

Anyone who has ever worked with user-submitted survey data has seen messy results like these.
Lots of things need to happen to make this list of strings uniform and ready for analysis: stripping whitespace, removing punctuation symbols, and standardizing on proper capitalization.
One way to do this is to use built-in string methods along with the re standard library module for regular expressions:

```
def remove_punctuation(value):
    return re.sub("[!#?]", "", value)

clean_ops = [str.strip, remove_punctuation, str.title]

def clean_strings(strings, ops):
    result = []
    for value in strings:
        for func in ops:
            value = func(value)
        result.append(value)
    return result

clean_strings(states, clean_ops)
```

```
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South   Carolina',
 'West Virginia']
```

```
for x in map(remove_punctuation, states):
    print(x)
```

```
   Alabama
Georgia
Georgia
georgia
FlOrIda
south   carolina
West virginia
```

Anonymous (Lambda) Functions

Python has support for so-called anonymous or lambda functions, which are a way of writing functions consisting of a single statement, the result of which is the return value. They are defined with the lambda keyword, which has no meaning other than "we are declaring an anonymous function":

```
def short_function(x):
    return x * 2

equiv_anon = lambda x: x * 2
```

- lambda functions are especially convenient in data analysis because, as you'll see, there are many cases where data transformation functions will take functions as arguments.
- It's often less typing (and clearer) to pass a lambda function as opposed to writing a full-out function declaration or even assigning the lambda function to a local variable. For example

```
def apply_to_list(some_list, f):
    return [f(x) for x in some_list]

ints = [4, 0, 1, 5, 6]
apply_to_list(ints, lambda x: x * 2)
```

```
[8, 0, 2, 10, 12]
```

```
strings = ["foo", "card", "bar", "aaaa", "abab"]

strings.sort(key=lambda x: len(set(x)))
strings
```

```
['aaaa', 'foo', 'abab', 'bar', 'card']
```

Generators

- Having a consistent way to iterate over sequences, like objects in a list or lines in a file, is an important Python feature.
- This is accomplished by means of the iterator protocol, a generic way to make objects iterable. For example, iterating over a dict yields the dict keys:

```
some_dict = {"a": 1, "b": 2, "c": 3}
for key in some_dict:
    print(key)
```

```
a
b
c
```

- An iterator is any object that will yield objects to the Python interpreter when used in a context like a for loop. Most methods expecting a list or list-like object will also accept any iterable object.

- This includes built-in methods such as min, max, and sum, and type constructors like list and tuple:

```
dict_iterator = iter(some_dict)
dict_iterator
```

```
<dict_keyiterator at 0x1aab56faac0>
```

```
list(dict_iterator)
```

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

- A generator is a convenient way, similar to writing a normal function, to construct a new iterable object.
- Whereas normal functions execute and return a single result at a time, generators return a sequence of multiple results lazily, pausing after each one until the next one is requested.
- To create a generator, use the yield keyword instead of return in a function:

```
def squares(n=10):
    print(f"Generating squares from 1 to {n ** 2}")
    for i in range(1, n + 1):
        yield i ** 2

gen = squares()
gen
```

```
<generator object squares at 0x000001AAB5606420>
```

```
for x in gen:
    print(x, end=" ")
```

```
Generating squares from 1 to 100
1 4 9 16 25 36 49 64 81 100
```

Generator expresssions

- Another way to make a generator is by using a generator expression.
- This is a generator analogue to list, dict, and set comprehensions.
- To create one, enclose what would otherwise be a list comprehension within parentheses instead of brackets:

```
gen = (x ** 2 for x in range(100))
gen
```

```
<generator object <genexpr> at 0x000001AAB2B9F030>
```

```
sum(x ** 2 for x in range(100))
dict((i, i ** 2) for i in range(5))
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

itertools module

- The standard library itertools module has a collection of generators for many common data algorithms.
  - For example, groupby takes any sequence and a function,
  - grouping consecutive elements in the sequence by return value of the function.

- Here's an example:

```
import itertools
def first_letter(x):
    return x[0]

names = ["Alan", "Adam", "Wes", "Will", "Albert", "Steven"]

for letter, names in itertools.groupby(names, first_letter):
    print(letter, list(names)) # names is a generator
```

```
A ['Alan', 'Adam']
W ['Wes', 'Will']
A ['Albert']
S ['Steven']
```

```
float("1.2345")
#float("something")
```

```
1.2345
```

```
def attempt_float(x):
    try:
        return float(x)
    except:
        return x

attempt_float("1.2345")
attempt_float("something")
```

```
'something'
```

```
float((1, 2))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[173], line 1
----> 1 float((1, 2))

TypeError: float() argument must be a string or a real number, not 'tuple'
```

```
def attempt_float(x):
    try:
        return float(x)
    except ValueError:
        return x

attempt_float((1, 2))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[176], line 1
----> 1 attempt_float((1, 2))

Cell In[174], line 3, in attempt_float(x)
      1 def attempt_float(x):
      2     try:
----> 3         return float(x)
      4     except ValueError:
      5         return x

TypeError: float() argument must be a string or a real number, not 'tuple'
```

```
def attempt_float(x):
    try:
        return float(x)
    except (TypeError, ValueError):
        return x

path = "segismundo.txt"
f = open(path, encoding="utf-8")

lines = [x.rstrip() for x in open(path, encoding="utf-8")]
lines
```

```
['Sueña el rico en su riqueza,',
 'que más cuidados le ofrece;',
 '',
 'sueña el pobre que padece',
 'su miseria y su pobreza;',
 '',
 'sueña el que a medrar empieza,',
 'sueña el que afana y pretende,',
 'sueña el que agravia y ofende,',
 '',
 'y en el mundo, en conclusión,',
 'todos sueñan lo que son,',
 'aunque ninguno lo entiende.']
```

```
f.close()

with open(path, encoding="utf-8") as f:
    lines = [x.rstrip() for x in f]

f1 = open(path)
f1.read(10)
f2 = open(path, mode="rb")  # Binary mode
f2.read(10)
```

```
b'Sue\xc3\xb1a el '
```

```
f1.tell()
f2.tell()
```

```
10
```

```
import sys
sys.getdefaultencoding()
```

```
'utf-8'
```

```
f1.seek(3)
f1.read(1)
f1.tell()
```

```
4
```

```
f1.close()
f2.close()

path

with open("tmp.txt", mode="w") as handle:
    handle.writelines(x for x in open(path) if len(x) > 1)

with open("tmp.txt") as f:
    lines = f.readlines()

lines
```

```
['SueÃ±a el rico en su riqueza,\n',
 'que mÃ¡s cuidados le ofrece;\n',
 'sueÃ±a el pobre que padece\n',
 'su miseria y su pobreza;\n',
 'sueÃ±a el que a medrar empieza,\n',
 'sueÃ±a el que afana y pretende,\n',
 'sueÃ±a el que agravia y ofende,\n',
 'y en el mundo, en conclusiÃ³n,\n',
 'todos sueÃ±an lo que son,\n',
 'aunque ninguno lo entiende.\n']
```

```
import os
os.remove("tmp.txt")

with open(path) as f:
    chars = f.read(10)

chars
len(chars)
```

```
10
```

```
with open(path, mode="rb") as f:
    data = f.read(10)

data
```

```
b'Sue\xc3\xb1a el '
```

```
data.decode("utf-8")
data[:4].decode("utf-8")
```

```
---------------------------------------------------------------------------
UnicodeDecodeError                        Traceback (most recent call last)
Cell In[210], line 2
      1 data.decode("utf-8")
----> 2 data[:4].decode("utf-8")

UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in position 3: unexpected end of data
```

```
sink_path = "sink.txt"
with open(path) as source:
    with open(sink_path, "x", encoding="iso-8859-1") as sink:
        sink.write(source.read())

with open(sink_path, encoding="iso-8859-1") as f:
    print(f.read(10))
```

```
SueÃ±a el
```

```
os.remove(sink_path)

f = open(path, encoding='utf-8')
f.read(5)
f.seek(4)
f.read(1)
f.close()
```

```
---------------------------------------------------------------------------
```

```
UnicodeDecodeError                        Traceback (most recent call
last)
Cell In[216], line 4
      2 f.read(5)
      3 f.seek(4)
----> 4 f.read(1)
      5 f.close()

File <frozen codecs>:322, in decode(self, input, final)

UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb1 in position
0: invalid start byte
```