STATS 507
Data Analysis in Python

Week3-2: Files and Dictionaries

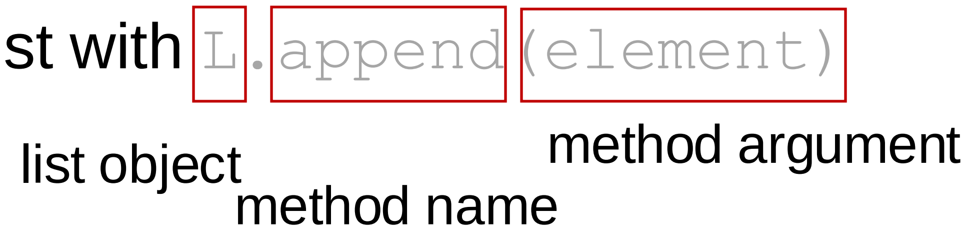
Dr. Xian Zhang

Recap: Lists in Python

Lists are (mutable) sequences whose values can be of **<u>any data type</u>**

- We call those list entries the **elements** of the list

Add an element to **<u>end</u>** of the list with



```
L.append(element)
```

list object

method name

method argument

Add an element to **<u>a specific location</u>** of the list with

```
L.insert(idx, element)
```

Add **multiple** element to the list.

Mutates the list and **return nothing**

```
L.extend(another_list)
```

Recap: Other lists operations in Python

Removes the first instance of x in the list by `list.remove(element)`.

`list.pop()` does two things:1) remove the last element from the list (mutate)

2) return that element

Sort a list.	`L.sort()` is a **method** associated with list and sorts the list **in place**. See documentation for how Python sorts data of different types: https://docs.python.org/3/howto/sorting.html
Reverse a list.	`L.reverse()`
Sort (Again)	`sorted(l)` returns a sorted version of a list, leaving its argument unchanged

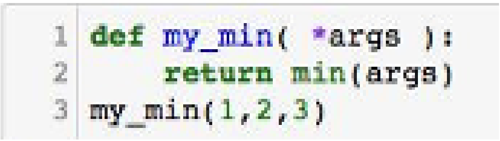
Conversion between strings and lists

Recap: Tuples in Python

Tuples are **immutable** sequences whose values can be of **any data type**

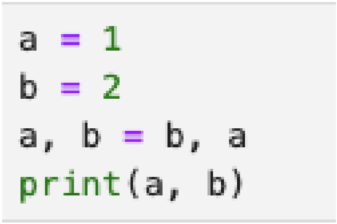
Create a tuple: tuples are created with “comma notation”, with optional parentheses:

- tuple_1 = 1, 2, 3
- tuple_2 = (2,3)
- tuple_3 = (2, "UM week 3", 98.0, True, [1,2,3])

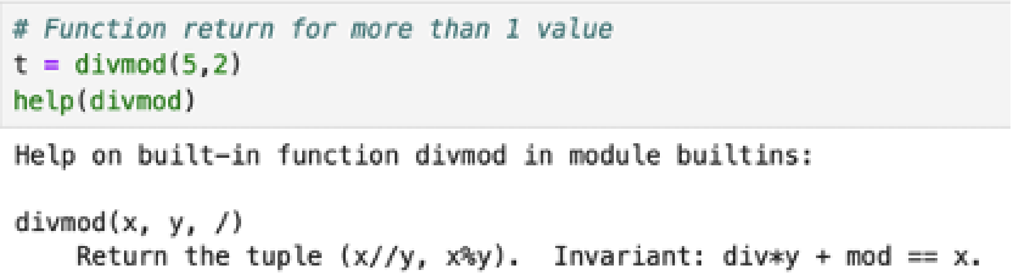


```
def my_min( *args ):
    return min(args)
my_min(1,2,3)
```

Applications:

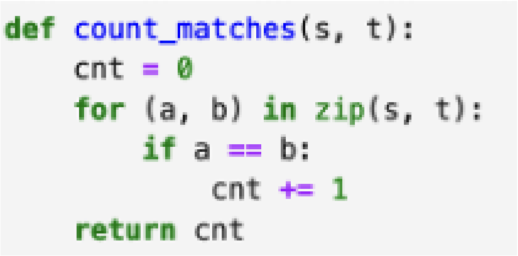


```
a = 1
b = 2
a, b = b, a
print(a, b)
```



```
# Function return for more than 1 value
t = divmod(5,2)
help(divmod)
```

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

divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.
```



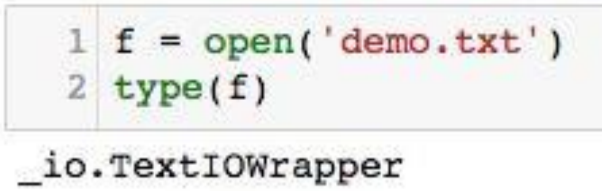
```
def count_matches(s, t):
    cnt = 0
    for (a, b) in zip(s, t):
        if a == b:
            cnt += 1
    return cnt
```

Recap: Files in Python

What are files?

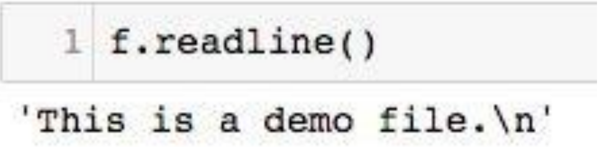
- Files are way to **store** and **manage** data on a computer.
- Also objects in Python

Create/operate a file object



```
f = open('demo.txt')
type(f)
```

```
_io.TextIOWrapper
```



```
f.readline()
```

```
'This is a demo file.\n'
```

```
# text-mode, read-only
open("readme.txt", "rt")
# text mode, write
open("readme.txt", "wt")
# text mode, append
open("readme.txt", "at")
# binary mode, read-only
open("data.dat", "rb")
# binary mode, write
open("data.dat", "wb")
# binary mode, append
open("data.dat", "ab")
```

1. More Files in Python

2. Dictionaries in Python

Formatting strings in Python

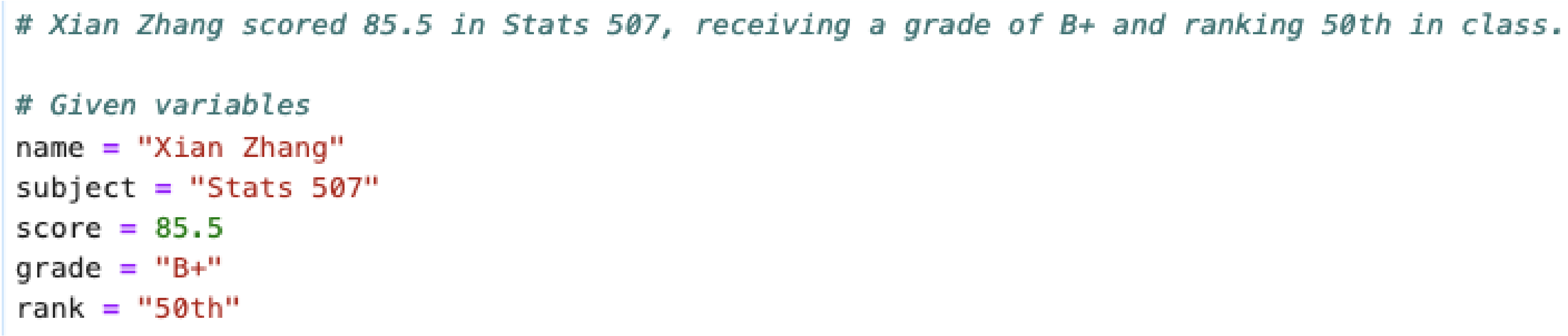
Very commonly, we want to write **formatted** string data to a file.

There are 3 ways of doing this in Python:

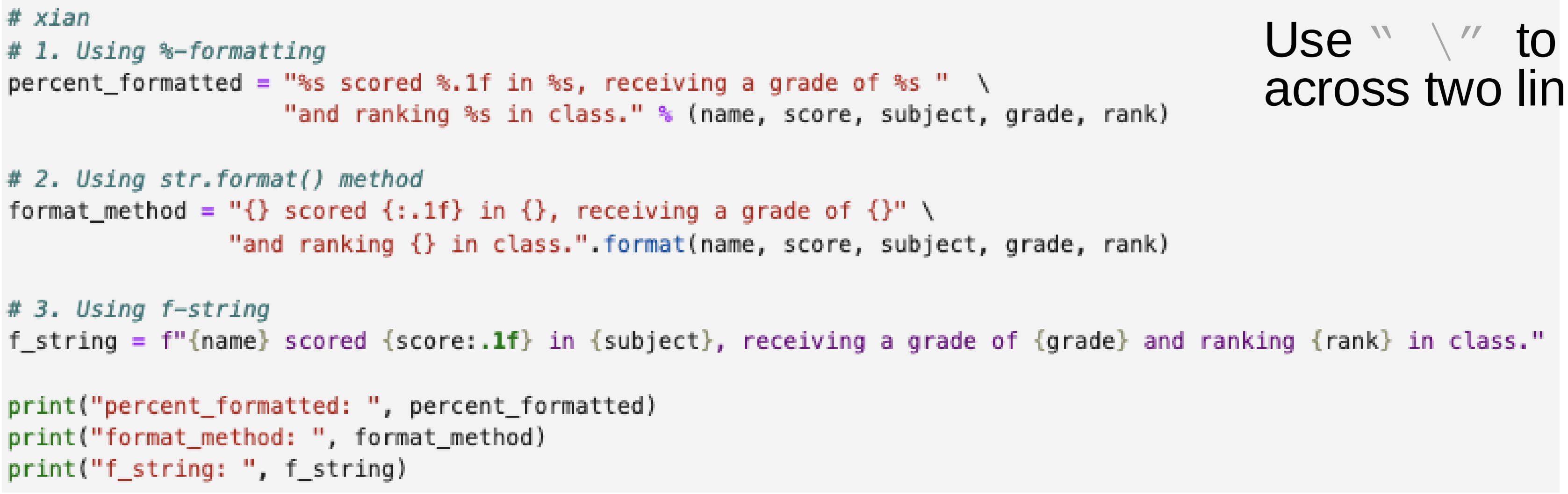
- The `%` operator (**old, avoid using this notation**)
- `string.format()`
- `f-strings (newest)`

```python
topping = "pineapple"

# all of these print
# "my fav pizza is pineapple"

"my fav pizza is %s" % topping

"my fav pizza is {}".format(topping)
"my fav pizza is {a}".format(a=topping)

f"my fav pizza is {topping}"
```

Formatting strings for practice



```
# Xian Zhang scored 85.5 in Stats 507, receiving a grade of B+ and ranking 50th in class.

# Given variables
name = "Xian Zhang"
subject = "Stats 507"
score = 85.5
grade = "B+"
rank = "50th"
```



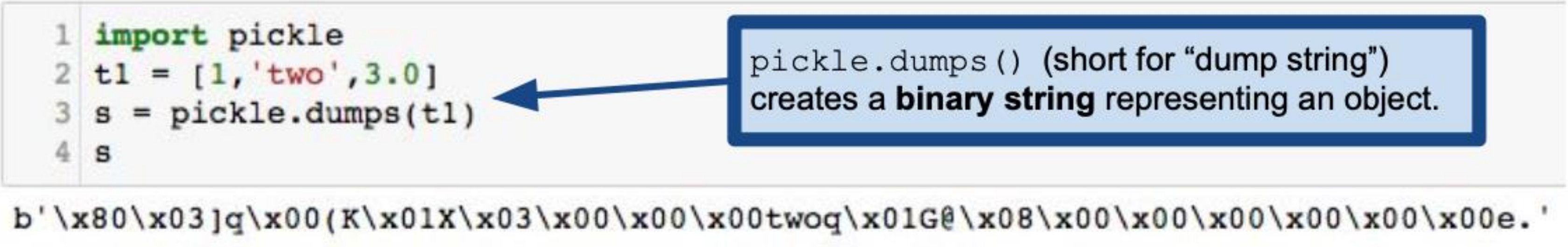
```
# xian
# 1. Using %-formatting
percent_formatted = "%s scored %.1f in %s, receiving a grade of %s "  \
                    "and ranking %s in class." % (name, score, subject, grade, rank)

# 2. Using str.format() method
format_method = "{} scored {:.1f} in {}, receiving a grade of {}" \
                "and ranking {} in class.".format(name, score, subject, grade, rank)

# 3. Using f-string
f_string = f"{name} scored {score:.1f} in {subject}, receiving a grade of {grade} and ranking {rank} in class."

print("percent_formatted: ", percent_formatted)
print("format_method: ", format_method)
print("f_string: ", f_string)
```

Use “ \” to split the string formatting across two lines for better readability

Just a bit more on files…

Saving objects as string to files…

Now we can write a string to a file. But not all object are strings…
Sometimes it is useful to be able to turn an object into a string.



```
import pickle
t1 = [1,'two',3.0]
s = pickle.dumps(t1)
s
```

```
b'\x80\x03]q\x00(K\x01X\x03\x00\x00\x00twoq\x01G@\x08\x00\x00\x00\x00\x00\x00e.'
```

`pickle.dumps()` (short for "dump string") creates a **binary string** representing an object.

This is a raw binary string that **encodes** the list t1. Each symbol encodes one byte.

`pickle` module in Python is used for **serializing** and **deserializing** Python objects.

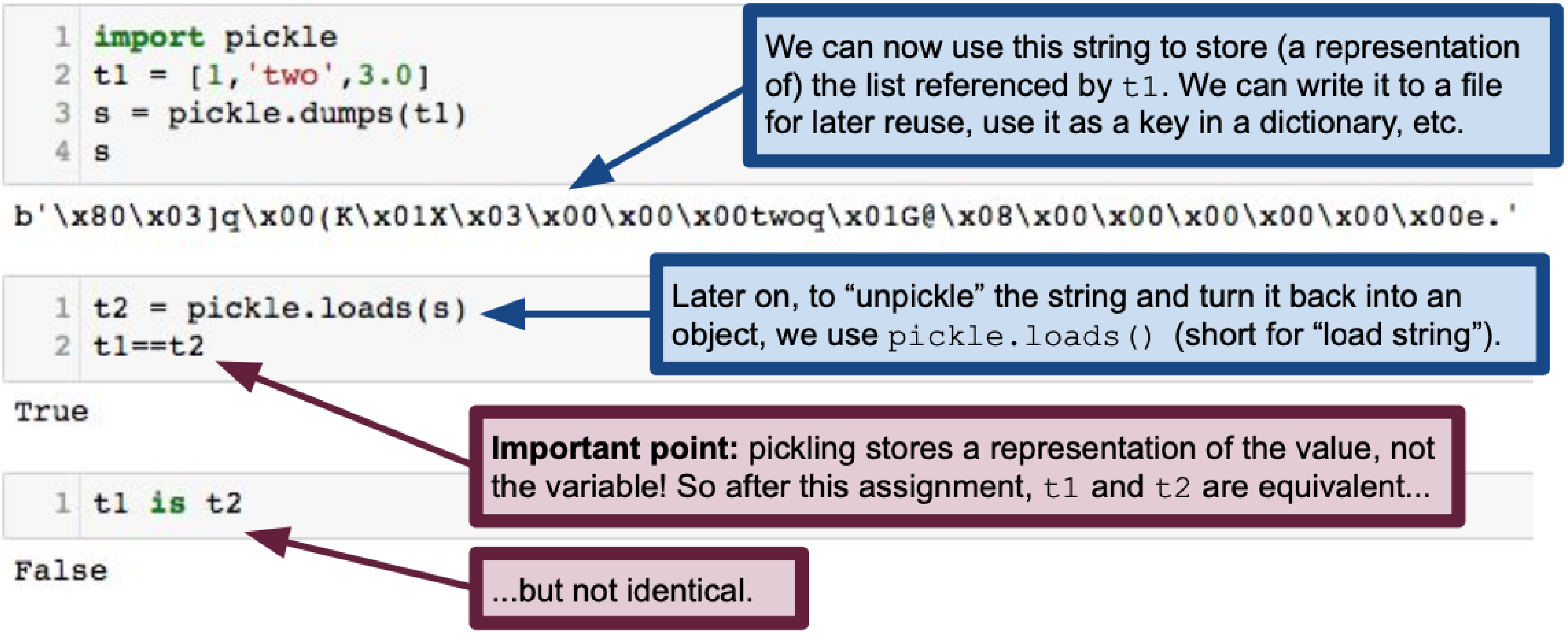
- Serialization: Convert Python objects into a byte stream.
- Deserialization: Convert a byte stream back into Python objects

https://docs.python.org/3/library/pickle.html https://docs.python.org/3/library/codecs.html

Saving (any) object to files…

`pickle` module in Python is used for serializing and deserializing Python objects.

- Serialization: Convert Python **<u>objects</u>** into a **<u>byte</u>** stream.
- Deserialization: Convert a byte stream back into Python objects



```python
import pickle
t1 = [1,'two',3.0]
s = pickle.dumps(t1)
s
```

```
b'\x80\x03]q\x00(K\x01X\x03\x00\x00\x00twoq\x01G@\x08\x00\x00\x00\x00\x00\x00e.'
```

We can now use this string to store (a representation of) the list referenced by `t1`. We can write it to a file for later reuse, use it as a key in a dictionary, etc.

```python
t2 = pickle.loads(s)
t1==t2
```

```
True
```

Later on, to “unpickle” the string and turn it back into an object, we use `pickle.loads()` (short for “load string”).

Important point: pickling stores a representation of the value, not the variable! So after this assignment, `t1` and `t2` are equivalent...

```python
t1 is t2
```

```
False
```

...but not identical.

Locating files in Python: the `os` module

`os` module lets us interact with the operating system:

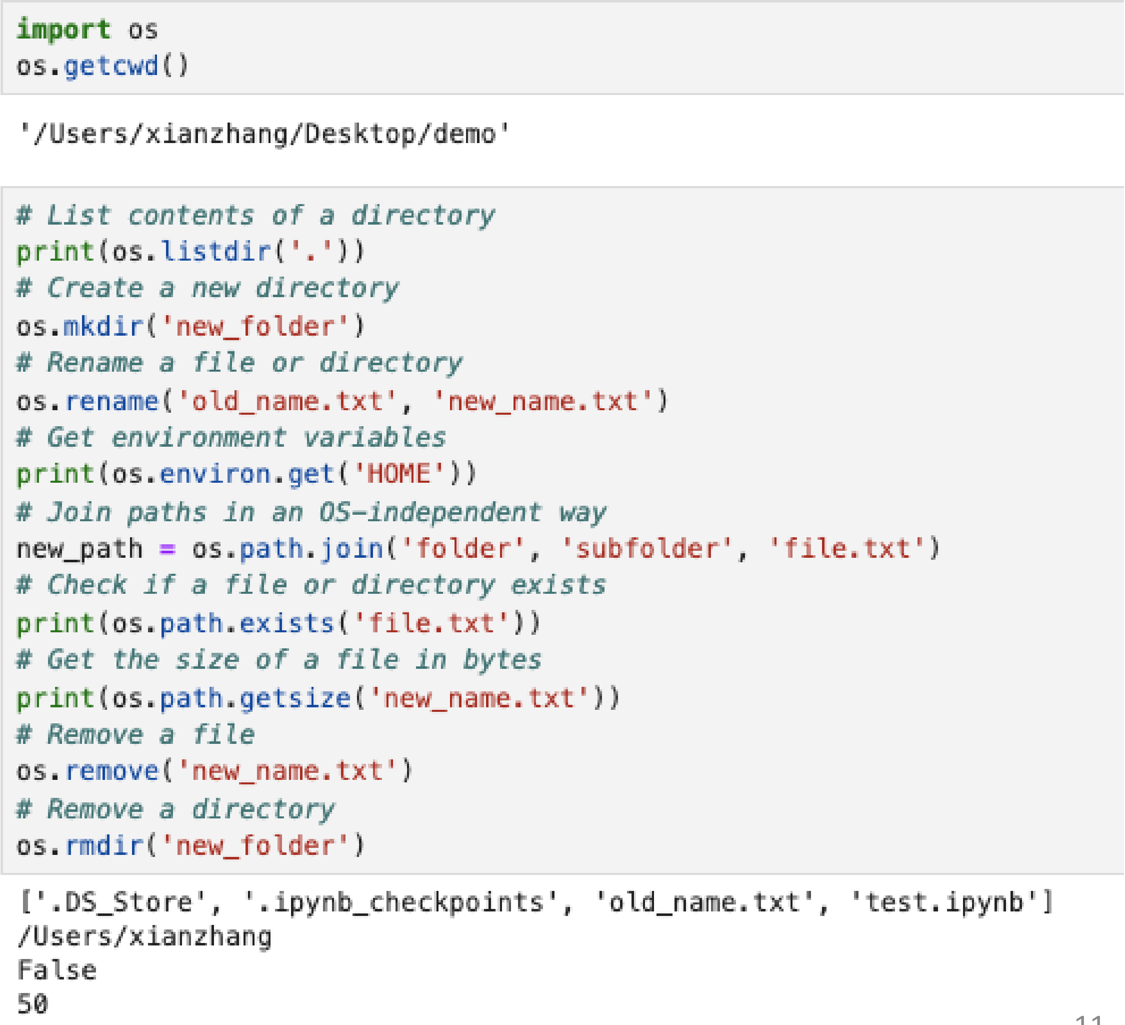
- `os.getcwd()` returns a string corresponding to the **current working directory**.

- `os.listdir()` lists the contents of its argument, or the current directory if no argument.

- A **path:** It starts at the **root directory**, ‘/’, and describes a sequence of nested directories

- A path from the root to a file or directory is called an **absolute path**. A path from the current directory is called a **relative path**.

- Read more: https://docs.python.org/3/library/os.html



```
import os
os.getcwd()
```

```
'/Users/xianzhang/Desktop/demo'
```

```
# List contents of a directory
print(os.listdir('.'))
# Create a new directory
os.mkdir('new_folder')
# Rename a file or directory
os.rename('old_name.txt', 'new_name.txt')
# Get environment variables
print(os.environ.get('HOME'))
# Join paths in an OS-independent way
new_path = os.path.join('folder', 'subfolder', 'file.txt')
# Check if a file or directory exists
print(os.path.exists('file.txt'))
# Get the size of a file in bytes
print(os.path.getsize('new_name.txt'))
# Remove a file
os.remove('new_name.txt')
# Remove a directory
os.rmdir('new_folder')
```

```
['.DS_Store', '.ipynb_checkpoints', 'old_name.txt', 'test.ipynb']
/Users/xianzhang
False
50
```

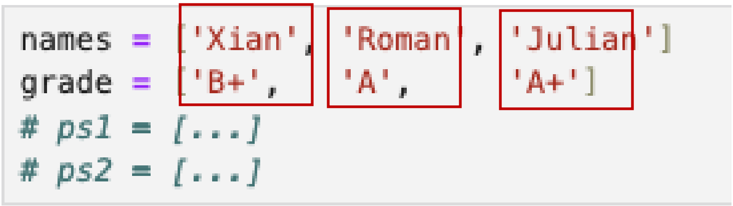
1. Files in Python

2. Dictionaries in Python

Why do we need yet another data type…

Suppose we want to store and use grade information for a set of students in Python.

- One way is to use list (mutable, add, delete, change…)



```
names = ['Xian', 'Roman', 'Julian']
grade = ['B+',   'A',     'A+']
# ps1 = [...]
# ps2 = [...]
```

- Info stored across many lists at the **same index**
- **Indirectly** access information by find locations..

Not effective, not efficient…
Solutions?

Dictionaries in Python

A Python dictionary has entries that map a **key: value**

A list

0	Elem1
1	Elem2
2	Elem3
3	Elem4
…	…

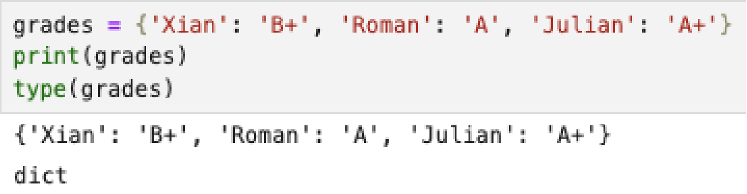
index element

A dictionary

key1	val1
key2	val2
key3	val3
key4	val4
…	…

Customized index value

A better and cleaner way to create grades -- use dictionary



```
grades = {'Xian': 'B+', 'Roman': 'A', 'Julian': 'A+'}
print(grades)
type(grades)
```

```
{'Xian': 'B+', 'Roman': 'A', 'Julian': 'A+'}
dict
```

Creating Dictionaries in Python

A Python dictionary store pairs of data as an entry

- key (customized index, any immutable object)
  - str, int, bool, tuple, **NOT** list, **NOT** dict
- value (can be any object)
  - all above plus lists and dicts!

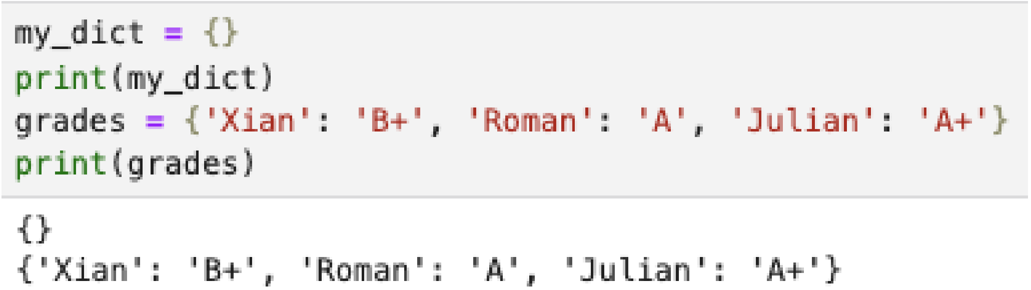
A dictionary

key1	val1
key2	val2
key3	val3
key4	val4
…	…

Customized index

value

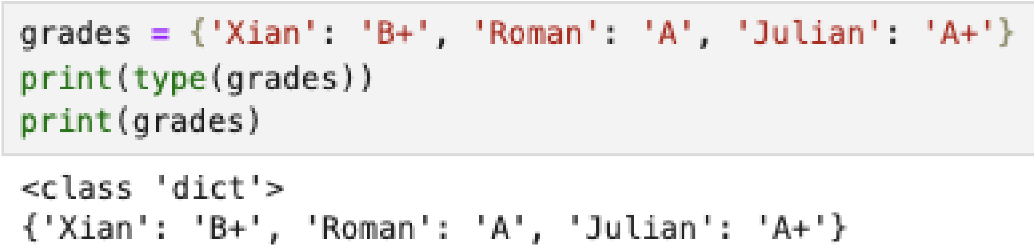
Use { } to create a dictionary, an entry maps a key to a value



```
my_dict = {}
print(my_dict)
grades = {'Xian': 'B+', 'Roman': 'A', 'Julian': 'A+'}
print(grades)
```

```
{}
{'Xian': 'B+', 'Roman': 'A', 'Julian': 'A+'}
```

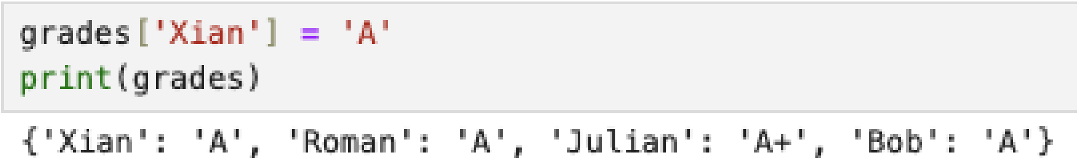
Dictionary operations in Python



```
grades = {'Xian': 'B+', 'Roman': 'A', 'Julian': 'A+'}
print(type(grades))
print(grades)
```

```
<class 'dict'>
{'Xian': 'B+', 'Roman': 'A', 'Julian': 'A+'}
```

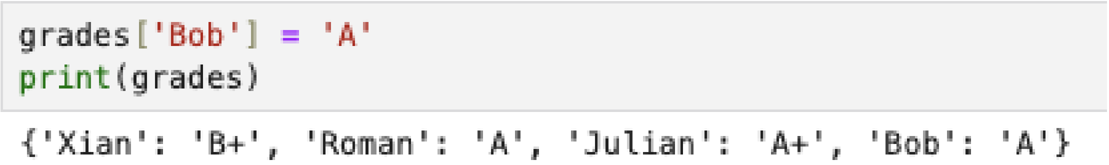
Modify/change an entry (**existing** key).



```
grades['Xian'] = 'A'
print(grades)
```

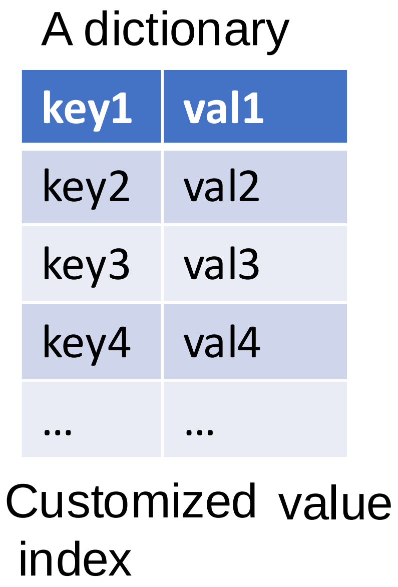
```
{'Xian': 'A', 'Roman': 'A', 'Julian': 'A+', 'Bob': 'A'}
```

Add an entry (add a key-value pair)



```
grades['Bob'] = 'A'
print(grades)
```

```
{'Xian': 'B+', 'Roman': 'A', 'Julian': 'A+', 'Bob': 'A'}
```



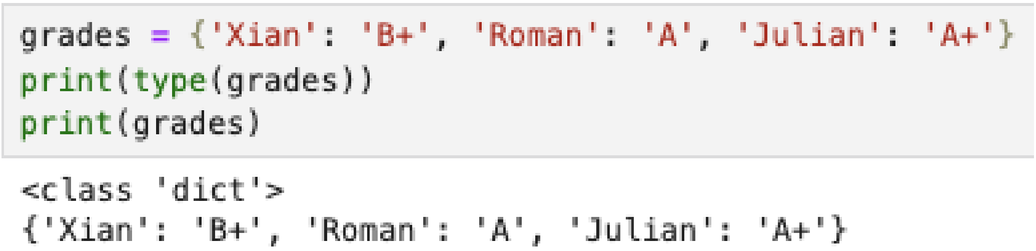
A dictionary

key1	**val1**
key2	val2
key3	val3
key4	val4
…	…

Customized index

value

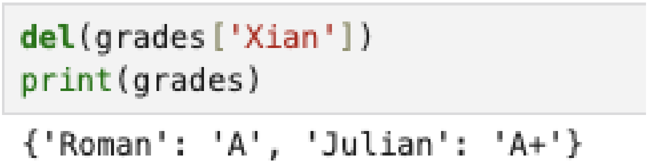
Dictionary operations: delete an entry



```
grades = {'Xian': 'B+', 'Roman': 'A', 'Julian': 'A+'}
print(type(grades))
print(grades)
```

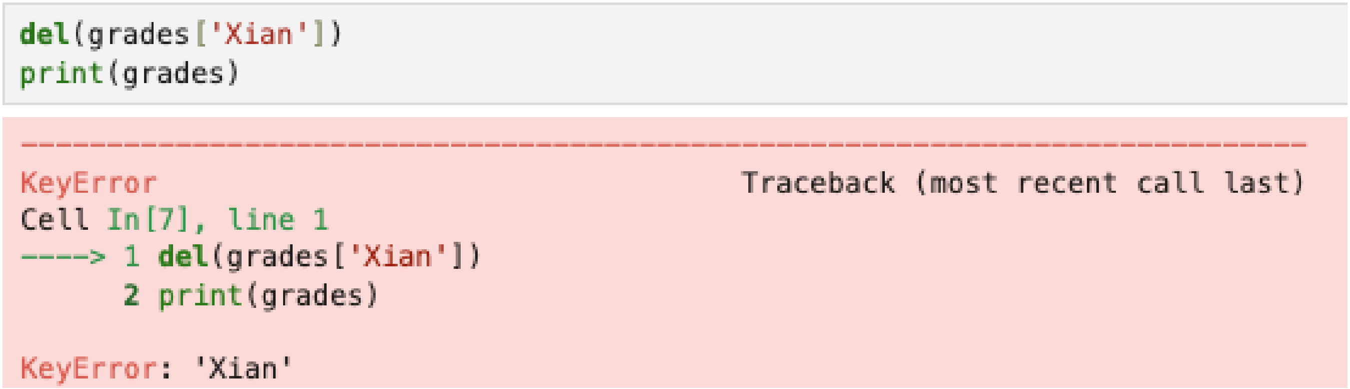
```
<class 'dict'>
{'Xian': 'B+', 'Roman': 'A', 'Julian': 'A+'}
```

Delete an entry using the `del()` function: mutates the dictionary



```
del(grades['Xian'])
print(grades)
```

```
{'Roman': 'A', 'Julian': 'A+'}
```

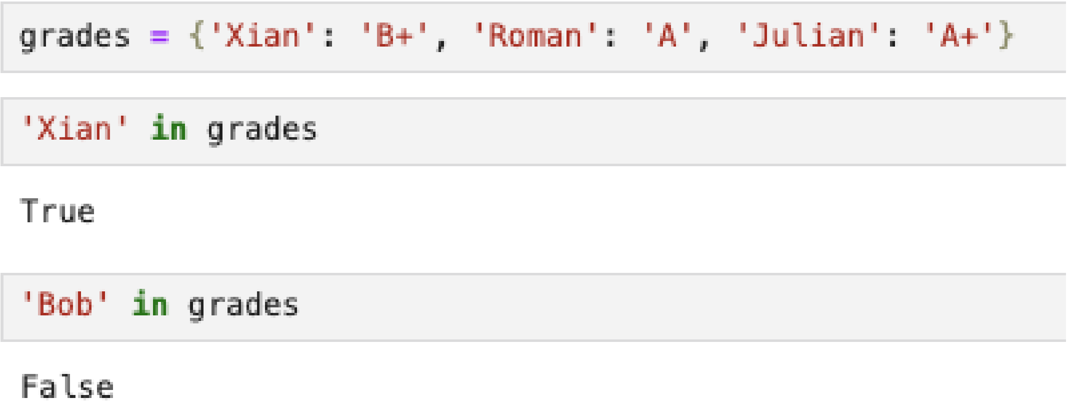


```
del(grades['Xian'])
print(grades)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[7], line 1
----> 1 del(grades['Xian'])
      2 print(grades)

KeyError: 'Xian'
```

Check membership

Check if a key is in dictionary using `in`



```
grades = {'Xian': 'B+', 'Roman': 'A', 'Julian': 'A+'}
```

```
'Xian' in grades
```

```
True
```

```
'Bob' in grades
```

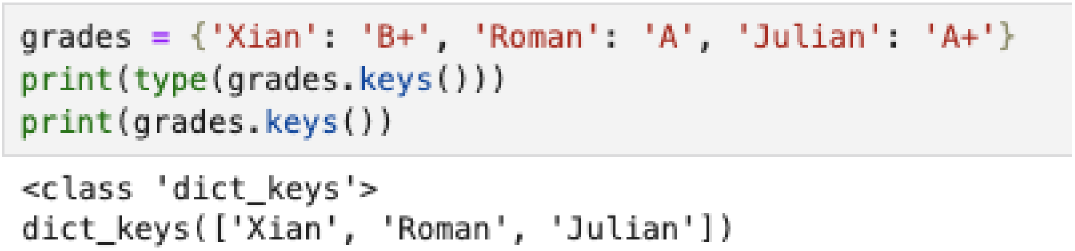
```
False
```

Note: `in` only checks the keys, NOT the values in the dictionaries.

Another Note: Dictionary for in check is much **faster** than list (more on this later!)

Iterate over dictionaries.

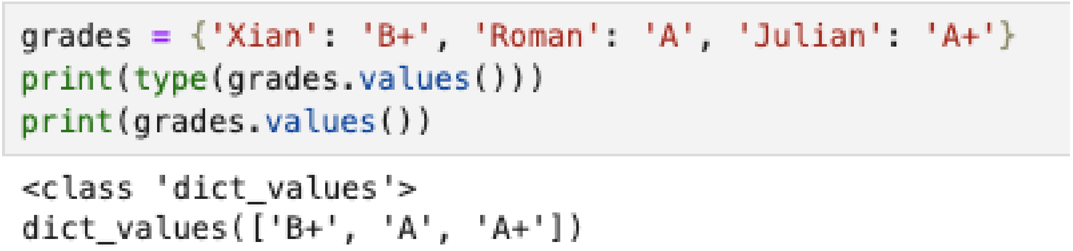
Get an **iterable** that acts like a tuple of all keys: `Dict.keys()`



```
grades = {'Xian': 'B+', 'Roman': 'A', 'Julian': 'A+'}
print(type(grades.keys()))
print(grades.keys())
```

```
<class 'dict_keys'>
dict_keys(['Xian', 'Roman', 'Julian'])
```

Get an **iterable** that acts like a tuple of all values: `Dict.values()`



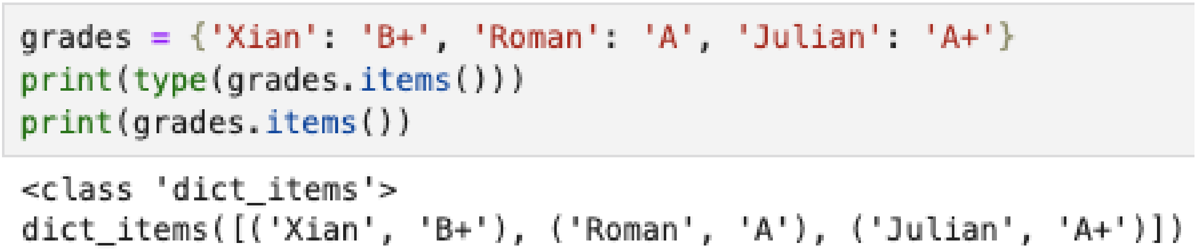
```
grades = {'Xian': 'B+', 'Roman': 'A', 'Julian': 'A+'}
print(type(grades.values()))
print(grades.values())
```

```
<class 'dict_values'>
dict_values(['B+', 'A', 'A+'])
```

Iterate over dictionaries.

Get an iterable that acts like a tuple of all items : `Dict.items()`

- To get the keys and values as pairs together



```
grades = {'Xian': 'B+', 'Roman': 'A', 'Julian': 'A+'}
print(type(grades.items()))
print(grades.items())
```

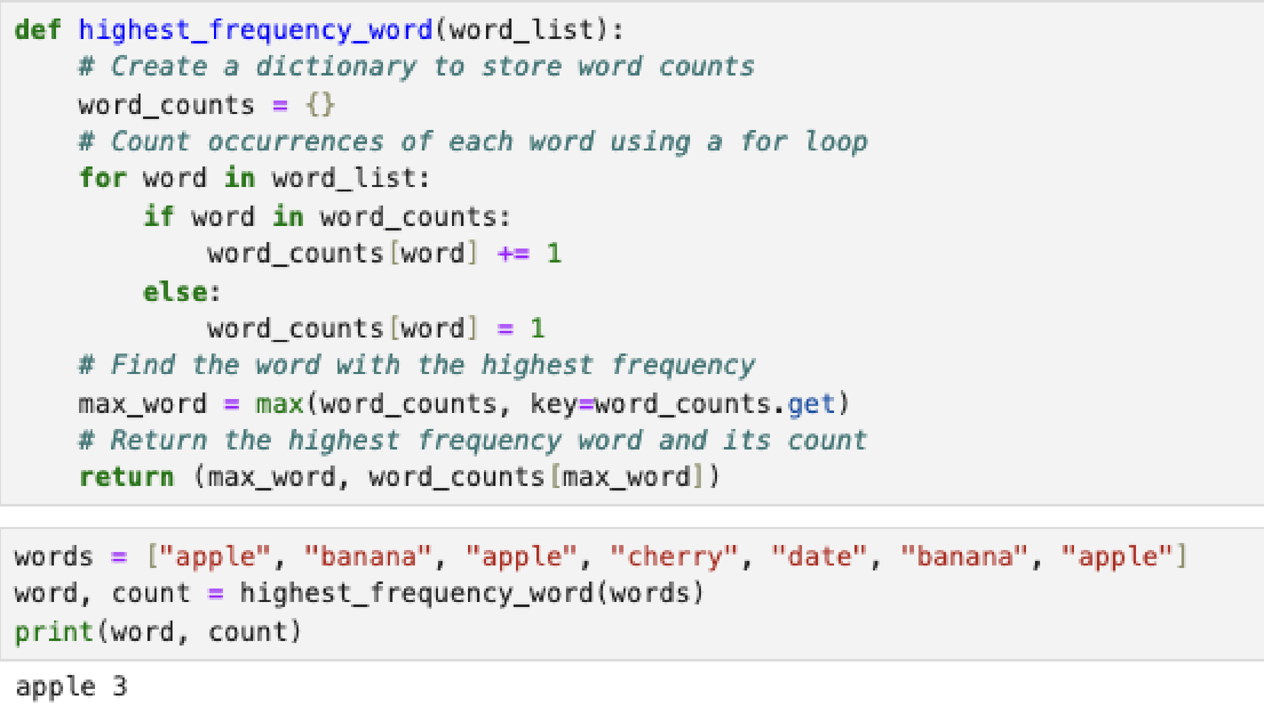
```
<class 'dict_items'>
dict_items([('Xian', 'B+'), ('Roman', 'A'), ('Julian', 'A+')])
```

In-class practice

Dictionary as a counter

We can use a dictionary a counter.

Example: counting words frequencies
the most frequent words in our favorite songs.



```
def highest_frequency_word(word_list):
    # Create a dictionary to store word counts
    word_counts = {}
    # Count occurrences of each word using a for loop
    for word in word_list:
        if word in word_counts:
            word_counts[word] += 1
        else:
            word_counts[word] = 1
    # Find the word with the highest frequency
    max_word = max(word_counts, key=word_counts.get)
    # Return the highest frequency word and its count
    return (max_word, word_counts[max_word])
```

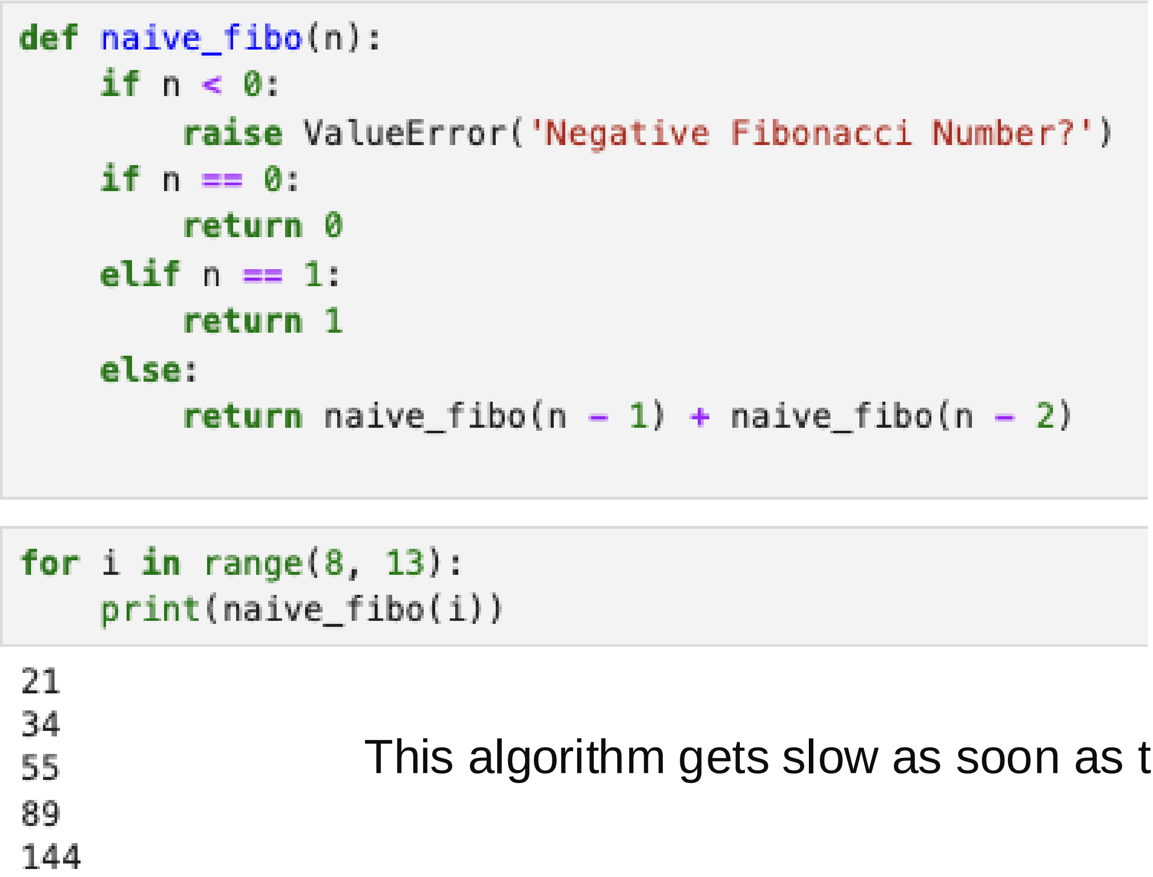
```
words = ["apple", "banana", "apple", "cherry", "date", "banana", "apple"]
word, count = highest_frequency_word(words)
print(word, count)
```

```
apple 3
```

Dictionary application: memoization

The Fibonacci Sequence

- 0,1,1,2,3,5,8,13,21…



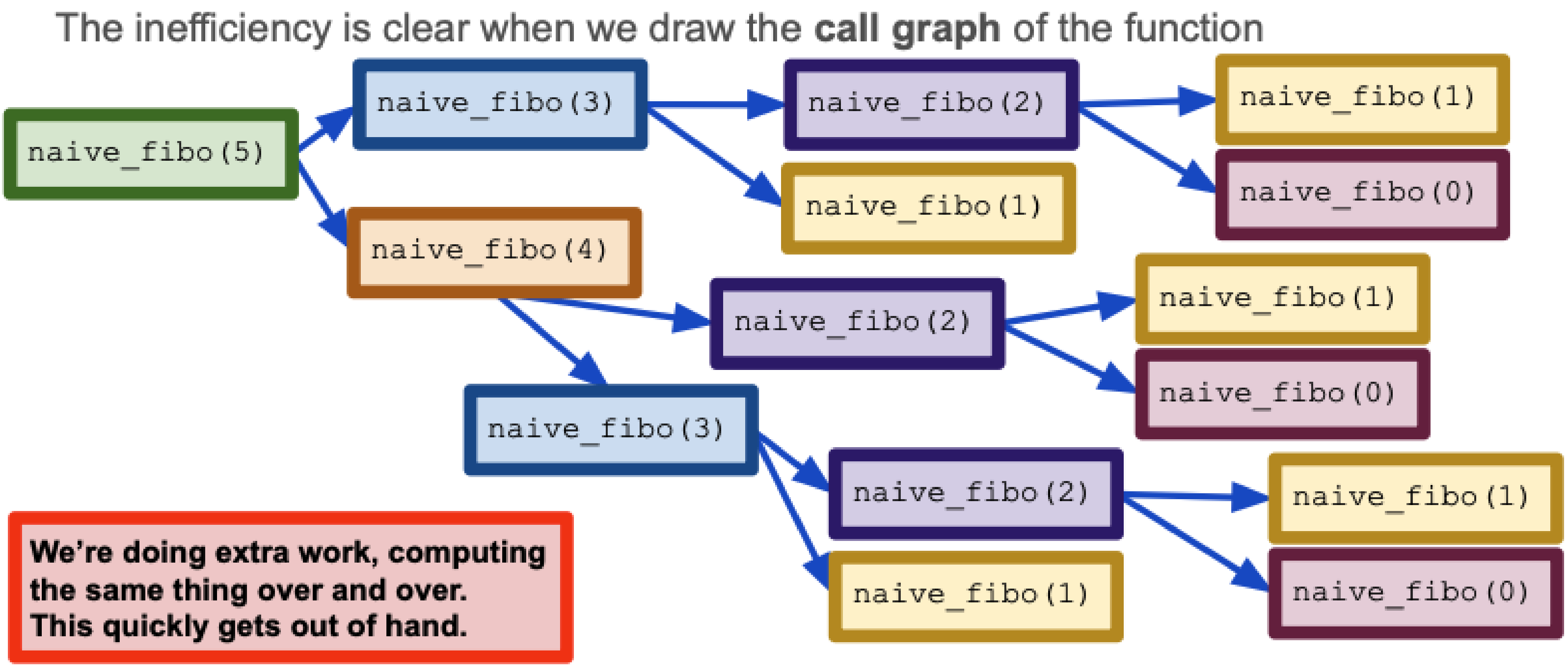
```
def naive_fibo(n):
    if n < 0:
        raise ValueError('Negative Fibonacci Number?')
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return naive_fibo(n - 1) + naive_fibo(n - 2)
```

```
for i in range(8, 13):
    print(naive_fibo(i))
```

```
21
34
55
89
144
```

This algorithm gets slow as soon as the argument get moderately big, why?

Dictionary application: memoization



The inefficiency is clear when we draw the **call graph** of the function

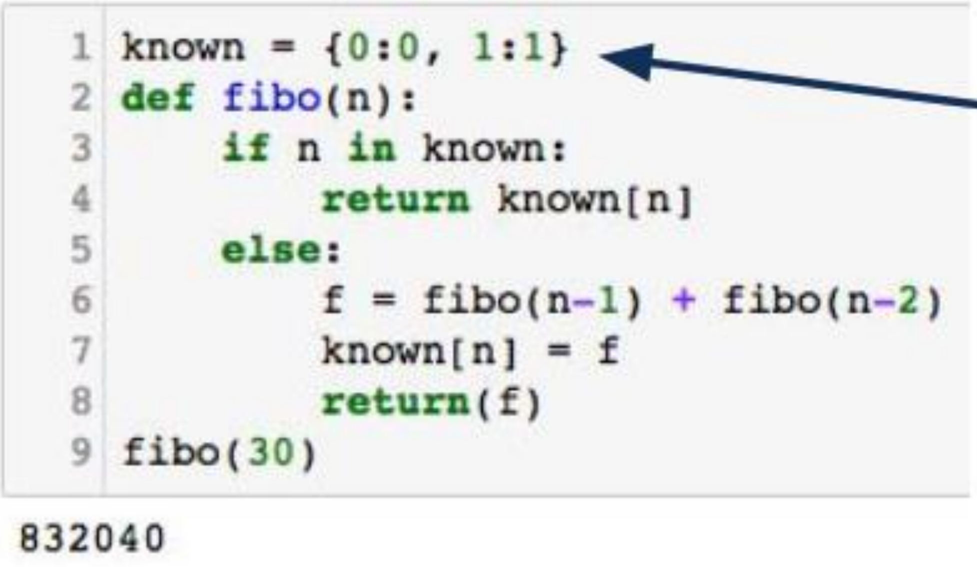
We're doing extra work, computing the same thing over and over. This quickly gets out of hand.

Let's visualize it in [Python tutor]

How to resolve this?

Store our computation for future **reuse**…

This is called **memorization**!



```
known = {0:0, 1:1}
def fibo(n):
    if n in known:
        return known[n]
    else:
        f = fibo(n-1) + fibo(n-2)
        known[n] = f
        return(f)
fibo(30)
```

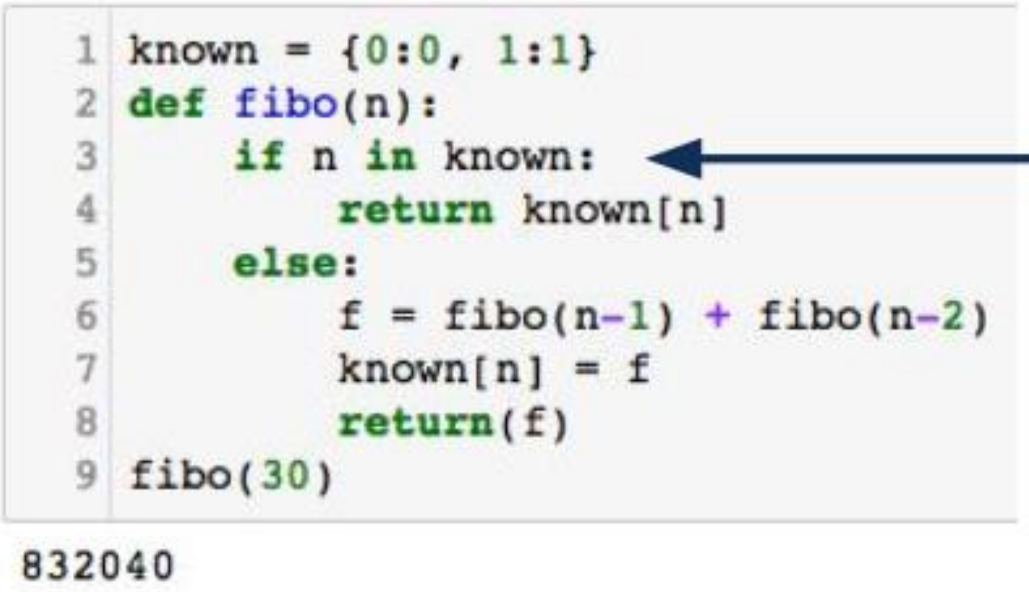
```
832040
```

This is the dictionary that we'll use for memoization. We'll store `known[n] = fibo(n)` the first time we compute `fibo(n)`, and every time we need it again, we just look it up!

Memoization

Store our computation for future **reuse**…

This is called **memorization**!



```
known = {0:0, 1:1}
def fibo(n):
    if n in known:
        return known[n]
    else:
        f = fibo(n-1) + fibo(n-2)
        known[n] = f
        return(f)
fibo(30)
```

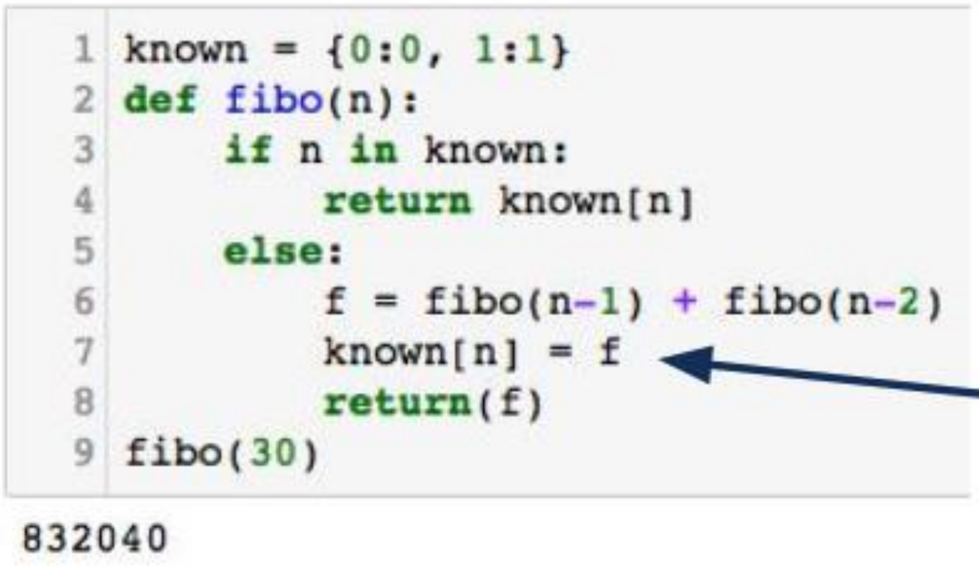
```
832040
```

If we already know the n-th Fibonacci number, there's no need to compute it again. Just look it up!

Memoization

Store our computation for future **reuse**…

This is called **memorization**!



```
known = {0:0, 1:1}
def fibo(n):
    if n in known:
        return known[n]
    else:
        f = fibo(n-1) + fibo(n-2)
        known[n] = f
        return(f)
fibo(30)
```

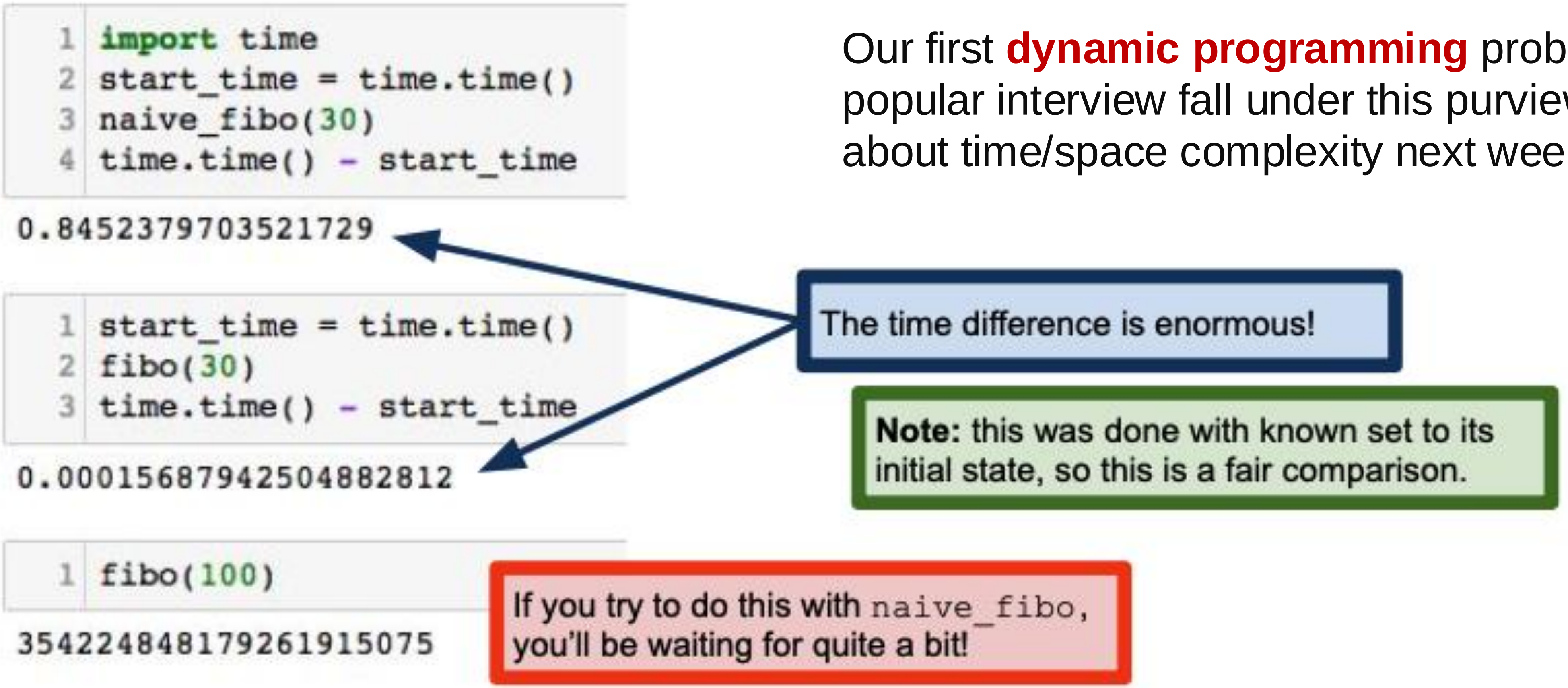
```
832040
```

If we don't already know it, we have to compute it, but before we return the result, we memoize it in `known` for future reuse.

Much more effective than naïve fibo

Using memorization is much more effective. Especially for huge numbers.

Our first **dynamic programming** problem, lots of popular interview fall under this purview, we will talk about time/space complexity next week!



```
import time
start_time = time.time()
naive_fibo(30)
time.time() - start_time
```

```
0.8452379703521729
```

```
start_time = time.time()
fibo(30)
time.time() - start_time
```

```
0.00015687942504882812
```

```
fibo(100)
```

```
354224848179261915075
```

The time difference is enormous!

Note: this was done with known set to its initial state, so this is a fair comparison.

If you try to do this with `naive_fibo`, you'll be waiting for quite a bit!

Summary on data types in Python: primitive data type

Different **object** can **represent** different concepts.

ANY object has a type that defines what kind of operations programs can do to them

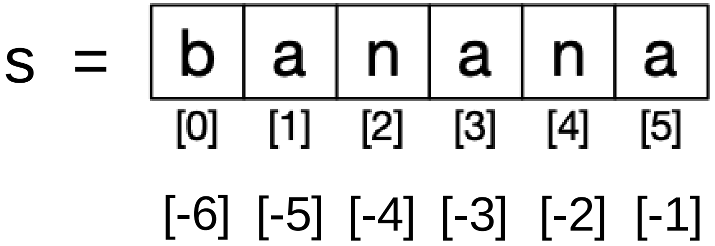
- int, -- represent integers, ex: 507
- float, -- represent real numbers, ex: 3.1415, 2.0
- bool, -- represent Boolean values, ex: True, False
- NoneType -- special and has one value, None

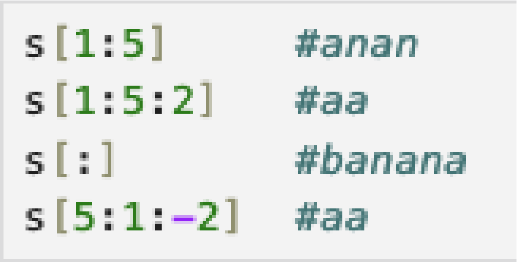
Mathematical operator: +, -, *, /, **, //, %

Logical operator: and, or, not

Summary on data types in Python: strings, lists and tuples

- Sequence: Indexing, slicing, `len()` …
- Immutability of string and tuple
- Mutable list (add, delete, reorder…)





```
s[1:5]       #anan
s[1:5:2]     #aa
s[:]         #banana
s[5:1:-2]    #aa
```

Summary on data types in Python: dictionaries

A Python dictionary has entries that map a **key: value**

A list

0	**Elem1**
1	Elem2
2	Elem3
3	Elem4
…	…

index element

A dictionary

key1	**val1**
key2	val2
key3	val3
key4	val4
…	…

Customized index value

Add, change, delete, iterate…

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

Where to train your muscles?

Practicing Python via:

https://www.hackerrank.com/

https://leetcode.com/

Coming next:

Part 1: Introduction to Python

Data types, functions, classes, objects, algorithm thinking, functional programming

Part 2: Numerical Computing and Data Visualization

numpy, scipy, scikit-learn, matplotlib, Seaborn

Part 3: Dealing with structured data

pandas, regular expressions, retrieving web data, SQL, real datasets

Part4: Intro to Deep Learning

PyTorch, Perceptron, Multi-layer perceptron, SGD, regularization, CNN…

Other things

HW2 due today.

HW3 out today.

Coming next:

Objects in Python (OOP), algorithm thinking and functional programming