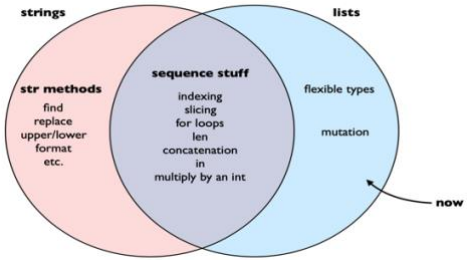
會考 exam1 要複習

LEC 15 List

lists are mutable

operation:

- Indexing
- Slicing
- For loops
- Len
- Concatenation
- In
- Multiply by an int
- 

Ways to mutate a list

- L[index] = new_value
- L.append(new_value)
- L.extend(another_list)
- L.pop(index)
- L.sort()
- sorted(L)

Transforming between String and List

- split( ) >> transforming a string to a list by the separator
  - List = S.split(“, ”)
- “ ”.join(L) >> transforming a list to a string by joining “”
  - S = “I”.join(List)

```
L = ["M", "SS", "SS", "PP", ""]
S = "I".join(L)
```

separator

what if removed?

>> MISSISSIPP

```
["M", "SS", "SS", "PP", ""]  ➡  MISSISSIPPI
```

Lec 16 CSV Tables

CSVs (Comma-Separated Values)

- Organize cells of data into rows and columns
- One sheet per file
- Only holds strings
- Delimiters
    - Act as separators
    - E.g. a new line delimits a row; a comma delimits between cells in a row

```
import csv

# copied from https://automatetheboringstuff.com/chapter14/
def process_csv(filename):
    exampleFile = open(filename, encoding="utf-8")
    exampleReader = csv.reader(exampleFile)
    exampleData = list(exampleReader)
    exampleFile.close()
    return exampleData
```

Lec 18 Dictionaries part 1

A **data structure** is a collection of data value[1], the relationships[2] among them and the functions or operations[3] that can be applied to the data

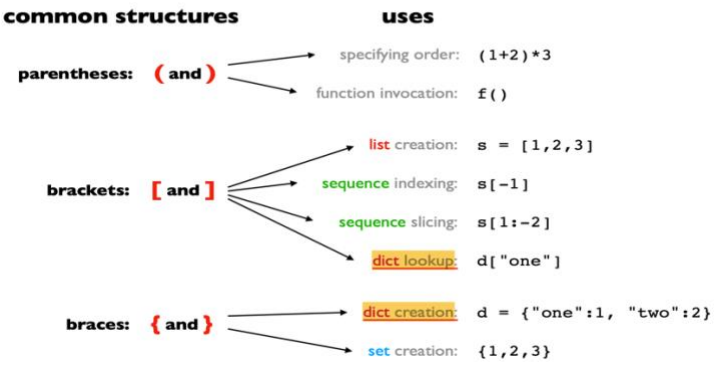
1. A list can contain a bunch of values of varying types
2. Every value has an index, representing an order within that list
3. L.sort(), len(L), L.pop(index), L.append(x), etc.

Lists are inflexible mapping structure because we don't have control over labels

Dictionary

- Dictionary uses **keys** to labels **values**
  - Keys must be immutable
  - Values can be anything

A note on parenthetical characters



-

- Mutations
  - Updates

```
>>> d = {0: "zero", 10: "ten", 20: "not set"}
>>> d[20] = "twenty"
>>> d
```

    - {0: 'zero', 20: 'twenty', 10: 'ten'}
  - Deletes

```
>>> d = {0: "zero", 10: "ten", 20: "not set"}
>>> d.pop(20)
'not set'
>>> d
```

    - {0: 'zero', 10: 'ten'}
  - Inserts

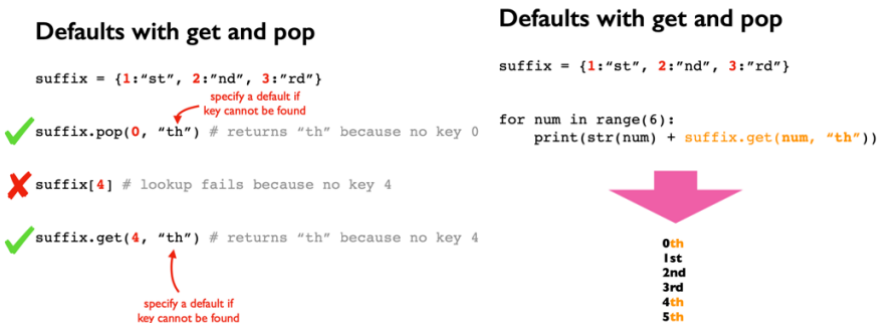
```
>>> d = {0: "zero", 10: "ten"}
>>> d[20] = "twenty"
>>> d
```

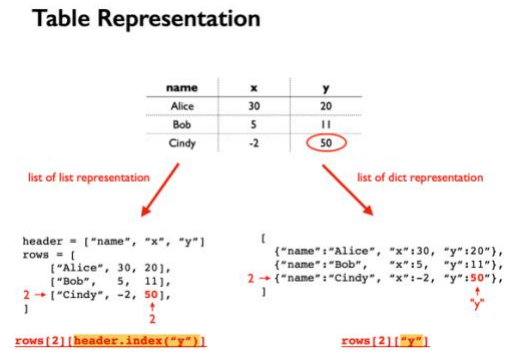
    - {0: 'zero', 20: 'twenty', 10: 'ten'}

Lec 19 Dictionaries part 2

- Creates an empty dict
  1. D = {} dict()
  2. D = ~~dict{}~~ (list: L = list(), set: S = set())
- Operations

num_words = {0:“zero”, 1:”one”, 2:”two”, 3:”three”}

  - len(num_words)) >> 4
  - 1 in num_words >> true
  - “one” in num_words >> false (only checks keys, not vals)
  - for x in num_words: >> iterates over keys (no orders)
  - list(num_words.keys()) >> return a list of keys
  - list(num_words.values()) >> return a list of vals
  - 

  - 

Lec 20 JSON

List of lists << -- >> CSV file
Dict of dicts << -- >> JSON file

JSON (JavaScript Object Notation):

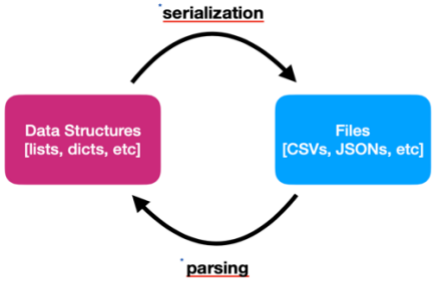
- Use curly braces
- Keys and vals are separated with a colon
- List use square brackets
- Strings are in quotes
- Integers look like integers

Minor JavaScript vs. Python differences:

will be on exam

	Python	JSON
Booleans	True, False	true, false
No value	None	null
Quotes	Single (') or double (")	Only double (")
Commas	Extra allowed: [1,2,]	No extra: [1,2]
Keys	Any type: {1: "one"}	Str only: {"1": "one"}

-



why not just have data structures?

because our data needs to live somewhere when our programs aren't running

why not just have files?

-

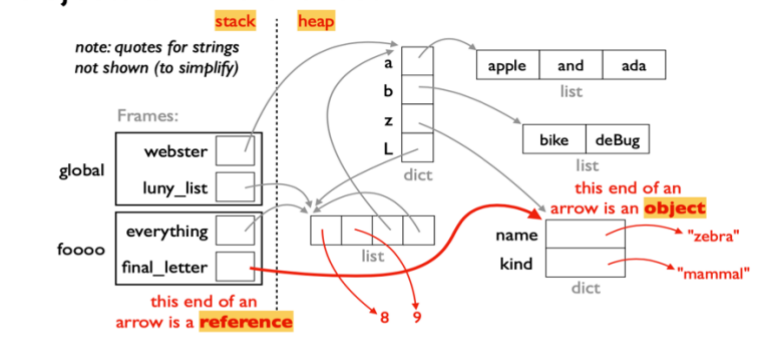
slow, and Python doesn't understand structure until it is parsed

Lec 21 Objects

mutable.
- list
- dict
- recordclass

immutable.
- tuple
- namedtuple

Objects and References



Observations
1. objects have a "life of their own" beyond variables or even function frames
2. here there are dict and list objects (others are possible)
3. references show up two places: as variables and values in data structures
4. technically ints and strs (and all values) are objects too in Python...

Tuple (immutable equivalent as list)

nums_list[0] = 200
nums_tuple[0] = 200

```
nums_list  = [200, 100, 300]
nums_tuple = (200, 100, 300)
```

- Tuples are immutable, unlike a list (mutable)

```
✔ nums_list[0] = 99
✘ nums_tuple[0] = 99
```

- Tuples can be used as keys

```
buildings = {
  (0,0): "Comp Sci",
  (0,2): "Psychology",
  (4,0): "Noland",
  (1,8): "Van Vleck"
}
```

Namedtuple (immutable)

```
people=[
  {"fname": "Alice", "lname": "Anderson", "age": 30},
  {"fname": "Bob", "lname": "Baker", "age": 31},
]
p = people[0]
print("Hello " + p["fname"] + " " + p["lname"])
```
dict

```
people=[
  ("Alice", "Anderson", 30),
  ("Bob", "Baker", 31),
]
p = people[1]
print("Hello " + p[0] + " " + p[1])
```
tuple

```
from collections import namedtuple
Person = namedtuple("Person", ["fname", "lname", "age"])
people=[
    Person("Alice", "Anderson", 30),
    Person("Bob", "Baker", 31),
]
p = people[0]
print("Hello " + p.fname + " " + p.lname)
```
namedtuple

```
from collections import namedtuple
```
need to import this data struct

```
Person = namedtuple("Person", ["fname", "lname", "age"])
```
name of that type; creates a new type!; name of that type

namedtuple → Person, Hurricane, ???? | number → int, float | sequence → str, list, tuple

```
p = Person("Alice", "Anderson", 30)
```
creates a object of type Person (sub type of namedtuple)
(like `str(3)` creates a new string or `list()` creates a new list)

```
print("Hello " + p.fname + " " + p.lname)
```

- Can either ~~use~~ use **positional** or **keyword** arguments to create a Person

Recordclass (support mutability, mutable, need additional install with "pip install record class")

```
from collections import namedtuple

Person = namedtuple("Person", ["fname", "lname", "age"])
p = Person(age=30, fname="Alice", lname="Anderson")

✘ p.age += 1 # it's a birthday!

print("Hello " + p.fname + " " + p.lname)
```
namedtuple

```
from recordclass import recordclass # not in collections!

Person = recordclass("Person", ["fname", "lname", "age"])
p = Person(age=30, fname="Alice", lname="Anderson")

✔ p.age += 1 # it's a birthday!

print("Hello " + p.fname + " " + p.lname)
```
recordclass

* the way namedtuple accessing data is the same as how recordclass accesses data.

Why does python have the complexity of separate references and objects?

1. Performance
    a. Easier to store reference to the actual object/data.
2. Centralized Updates

```
from recordclass import recordclass

Person = recordclass("Person", ["name", "score", "age"])

alice = Person(name="Alice", score=10, age=30)
bob = Person(name="Bob", score=8, age=25)
winner = alice

alice.age += 1
print("Winner age:", winner.age)
```

Example 1 – 5

```
def f(x):
  x *= 3
  print("f:", x)

num = 10
f(num)
print("after:", num)
```

integers are immutable, strings are immutable

Output:
f: 30
after: 10

```
def f(items):
  items.append("!!!")
  print("f:", items)

words = ['hello', 'world']
f(words)
print("after:", words)
```

if switch to

```
def f(items):
  items = items + ["!!!"]
  print("f:", items)

words = ['hello', 'world']
f(words)
print("after:", words)
```

lists are mutable

output:
["hello", "world", "!!!"]
["hello", "world", "!!!"]

output:
["hello", "world", "!!!"]
["hello", "world"]

```
def first(items):
    return items[0]

def smallest(items):
    items.sort()
    return items[0]

numbers = [4,5,3,2,1]
print("first:", first(numbers))
print("smallest:", smallest(numbers))
print("first:", first(numbers))
```

The primary difference between the list sort() function and the sorted() function is that the sort() function will modify the list it is called on. The sorted() function will create a new list containing a sorted version of the list it is given. The sorted() function will not modify the list passed as a parameter. If you want to sort a list but still have the original unsorted version, then you would use the sorted() function. If maintaining the original order of the list is unimportant, then you can call the sort() function on the list.

Output:
first: 4
smallest: 1
first: 1

```
def first(items):
    return items[0]

def smallest(items):
    items = sorted(items)
    return items[0]

numbers = [4,5,3,2,1]
print("first:", first(numbers))
print("smallest:", smallest(numbers))
print("first:", first(numbers))
```

Output:
first: 4
smallest: 1
first: 4

Is vs "=="

- == : check if two references refer to two equivalent objects
- Is: check if two references refer to the same object

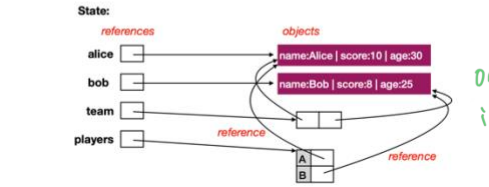
Lec 22 Copying

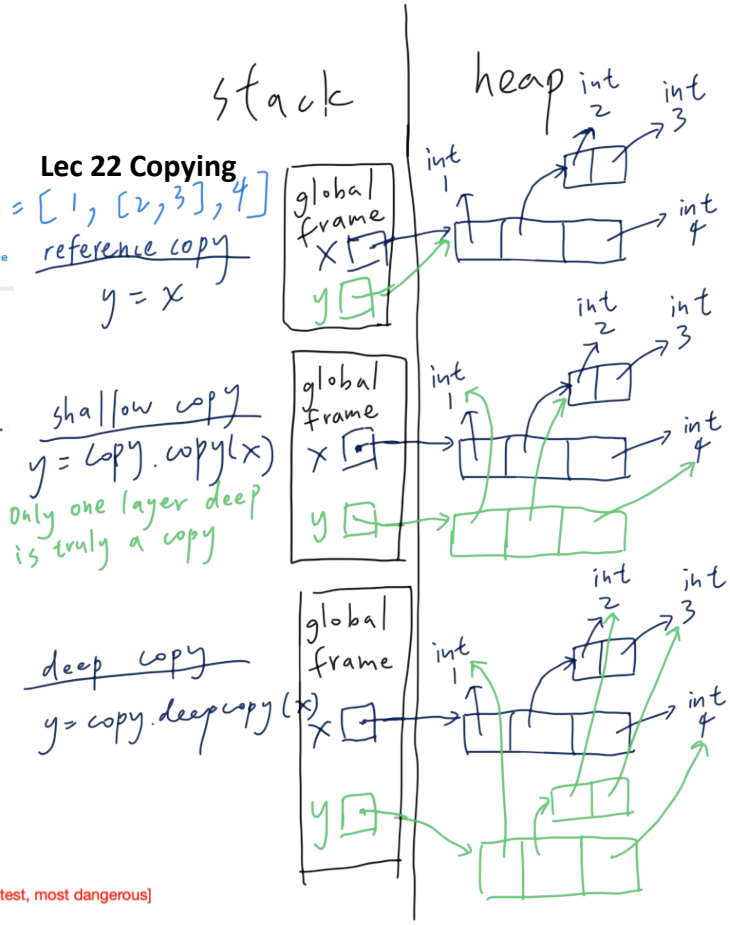
Reference:

what DID NOT happen: team contains the alice and bob variables

what DID happen: team contains references to the objects referenced by bob and alice

```
from recordclass import recordclass

Person = recordclass("Person", ["name", "score", "age"])

alice = Person(name="Alice", score=10, age=30)
bob = Person(name="Bob", score=8, age=25)
team = [alice, bob]
players = {"A": alice, "B": bob}
```





The levels of copy

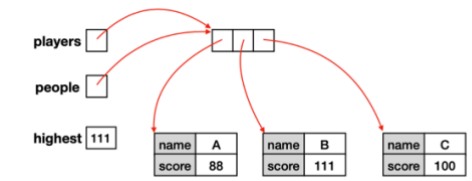
```
import copy
x = [
  {"name":"A", "score":88},
  {"name":"B", "score":111},
  {"name":"C", "score":100}
]

# uncomment one of these
#y = x                    <- reference copy [fastest, most dangerous]
#y = copy.copy(x)         <- shallow copy
#y = copy.deepcopy(x)     <- deep copy [slowest, safest]
```

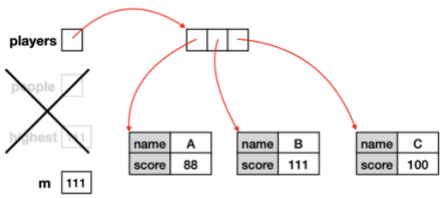
module function

y = x (reference copy)

```
def max_score(people):
  highest = None
  for p in people:
    if highest == None or p["score"] > highest:
      highest = p["score"]
  return highest

players = …
m = max_score(players)
```

```
def max_score(people):
  highest = None
  for p in people:
    if highest == None or p["score"] > highest:
      highest = p["score"]
  return highest

players = …
m = max_score(players)
```





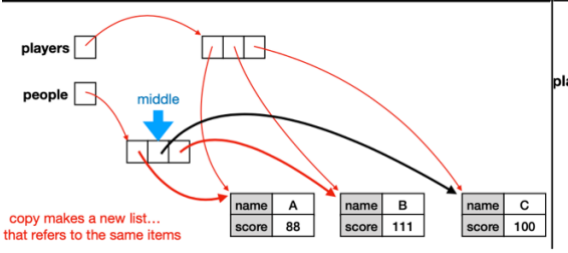
y = copy.copy(x) (shallow copy) (does not change the element when modifying y)

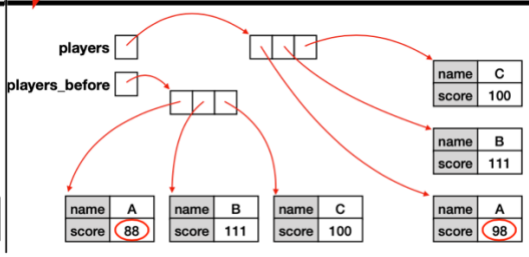
y = copy.deepcopy(x) (deep copy) (does not change the element when modifying y)

```
def median_score(people):
  people = copy.copy(people)
  people.sort(...)
  # TODO: return score for middle of people

players = …
m = median_score(players)
```

```
players = …
players_before = copy.deepcopy(players)

# make changes to players
players[0]["score"] += 10

print("score change:",
      players[0]["score"] - players_before[0]["score"])
```

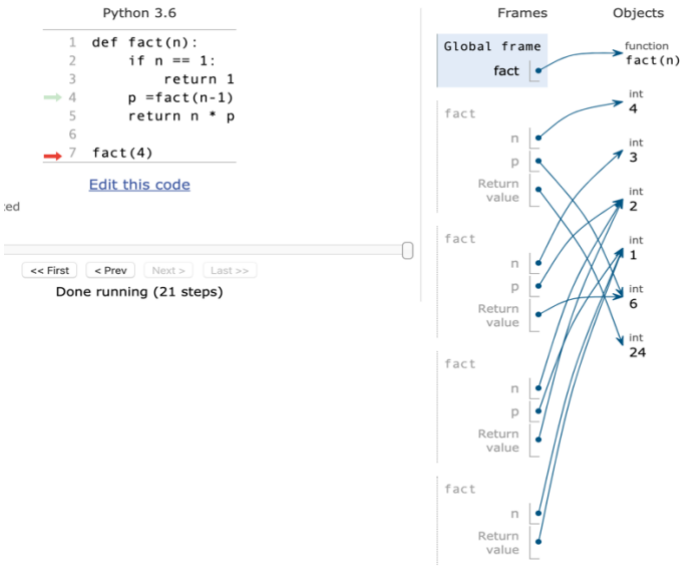
prints 10





Lec 23 Recursive Function

- Definition:
  - Wooden stick with an end splitting into other branches or terminating with a leaf.
    - Trees are arbitrarily large; recursive case allows indefinite growth
    - Trees are finite; eventual base case allows completion
  - In recursion, each function invocation has its **own state**, but multiple invocations **share code**.



  - Where do computers keep local variable for recursive calls?
    - In a section of memory called a “frame”
    - Only one function is active at a time, so keep frames in a stack
  - Infinite recursion
    - Calls keep pushing more frames
    - Exhaust memory, throw **StackOverflowError**

In recursion, there are multiple frames of the same function object in a stack.

Lec 24 Function Reference

Functions are objects

- Variables can reference functions (we can pass function references to other functions)
- Lists/Dicts can reference functions (we can pass lists of function references to other functions)

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

def f():
    return "hi"

g = f

z = f()
```

Explanation: x should reference a new list object

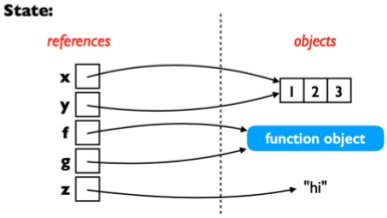
Explanation: y should reference whatever x references

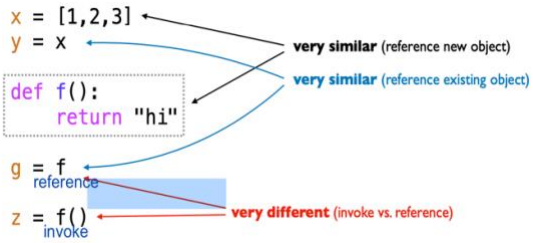
Explanation: f should reference a new function object

Explanation: g should reference whatever f references

Explanation: z should reference whatever f returns

both of these calls would have run the same code, returning the same result:
- `z = f()`
- `z = g()`





List of tuples

- ~~Sorst()~~ Sorting tuples.

```
names = [
    ("Catherine", "Baker"),
    ("Alice", "Clark"),
    ("Bob", "Adams"),
]

names.sort()
```

Catherine	Baker
Bob	Adams
Alice	Clark

sorting tuples is done on first element (ties go to 2nd element)

Alice	Clark
Bob	Adams
Catherine	Baker

- Sorted by last name

```
sorted by last name
def extract(name_tuple):
    return name_tuple[1]

names.sort(key=extract)
```

Sorted by length

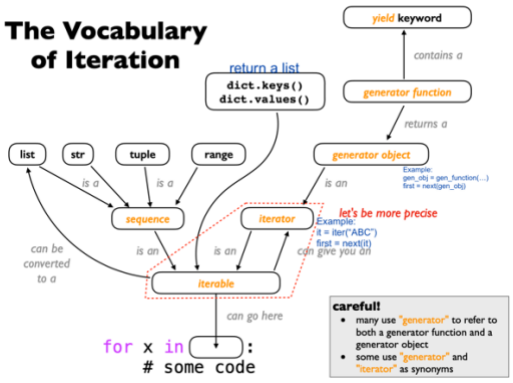
```
sorted by length
def extract(name_tuple):
    return len(name_tuple[0])

names.sort(key=extract)
```

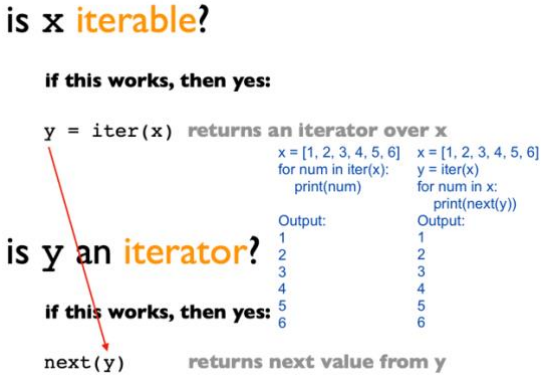
Iterators/Genertors

Genreators

- Any function containing the yield keyword anywhere is a generator

Vocabulary of interation





Can you classify x, y, and z?

1
```
>>> x = [1,2,3]
>>> next(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' object is not an iterator
>>> iter(x)
<list_iterator object at 0x1067bfb38>
```

2
```
>>> y = enumerate([1,2,3])
>>> next(y)
(0, 1)
```
=> returns a tuple
0 is just a counter

3
```
>>> z = 3
>>> next(z)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not an iterator
>>> iter(z)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

The Open Function

path = “file.txt”
f = open(path)

- path: path takes a string argument, which contains path to a file
- f: f returns a file object; **file objects are iterators**

Reading Files

```
path = "file.txt"
f = open(path)

for line in f:
    print(line)
```

file.txt

```
This is a test!
3
2
1
Go!
```

Output

```
This is a test!

3

2

1

Go!
```

-

generator function: Any function contains yield statement

generator object: A function with a yield statement will always return a generator object, regardless of what those yields are doing.

next( ) : type(yield)

Lec 26 Error Handling

Assert

- Syntax:
  - assert BOOLEAN_EXPRESSION
- Purpose:
  - Force program to crash if something is non-sensible (semantic errors are hard to debug), rather than run and produce wrong output

Try/Except

- If we want to keep running the program even if there is an error
  - try:
    some code here
    except:
    print("error caught")
  - Control flow

```
def buggy():
    print("buggy: about to fail")
    print("buggy: infinity is ", 1/0)
    print("buggy: oops!") # never prints

def g():
    print("g: before buggy")
    try:
        buggy()
    except:
        print("g: caught an exception from buggy")
    print("g: after buggy")

def f():
    try:
        print("f: let's call g")
        g()
        print("f: g returned normally")
    except:
        print("f: that didn't go so well")

f()
```

g catches, so f never knows about the exception

Output:
f: let's call g
g: before buggy
buggy: about to fail
g: caught an exception from buggy
g: after buggy
f: g returned normally.

- If we want to know the reason for the exception (**Catch Cause**)
  - try:
    some code here
    Except Exception as e:
    print("error message: ", str(e))

→ division by zero

- if we only want to catch certain exceptions (**Narrow Catching**)
  - try:
    some code here
    except (ValueError, IndexError) as e:
    print("error message: ", str(e))
- if we want to produce a specific kind of error

Asserts vs. Raising Exception Objects

Version 1 (quick and dirty):

Pizza Example:
- raise TypeError

```
def pizza_size(radius):
    assert type(radius) in (float, int)
    return (radius ** 2) * math.pi
```

Version 2 (more robust and informative):

```
def pizza_size(radius):
    if type(radius) not in (float, int):
        raise TypeError("need a numeric type")
    return (radius ** 2) * math.pi
```

tell Python this exception occurred here

create TypeError object

with this message

  -

Lec 27 Files and Directories

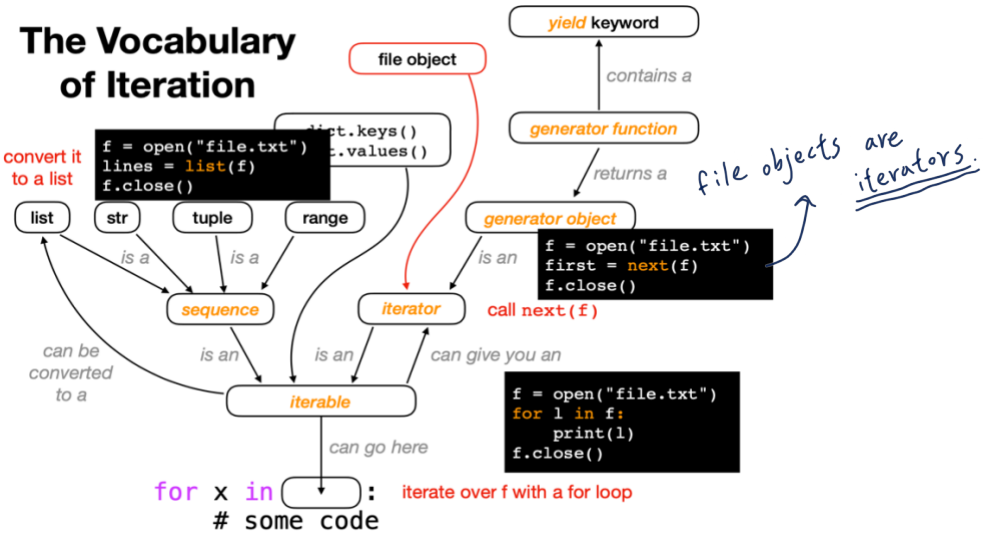
File

- f = open(path)

  # read data from f OR write data to f

  f.close()
  - f: file object
  - open: built in open function
  - path: file path (ex: file.txt; data/movies.csv; data/snapshots/A, etc.)
- f = open("file.txt")
  data = f.read()
  print(data)
  - option 1. read() method
    - fetch entire file contents and return as a string
  - option 2. File objects are iterators

when opening a file for reading, if the file doesn't exist, an error occurs.



- f = open("file.txt", "w") ("w" mode indicates we want to write to this file)
  f.write("hello")
  f.write(" world\n")
  f.write("!!!!\n")
  - opening file with "w" will immediately wipes out the file (or create a new one if there isn't already a "file.txt")

overwrite

OS module (must import os)

- os.listdir
  - list the files and directories
- os.mkdir
  - create a directory
- os.path.exists
  - return true if the file exists, false otherwise
- os.path.isfile
  - return true if the file could be found and **is a file**, false otherwise.
- or.path.isdir
  - return true if it **is a directory**, false otherwise
- os.path.join
  - >> import os
    >>os.path.join("data", "movies.csv")
    data/movies.csv

File exceptions

- Working with files leads to many exceptions
  - Missing files
  - Lacking permissions
  - Not enough space
  - Mixing up directories and files
  - Corrupt formats
  - etc., etc.
- try:
    some code
  except FileExistsError:
    pass

  some code
  …
  …