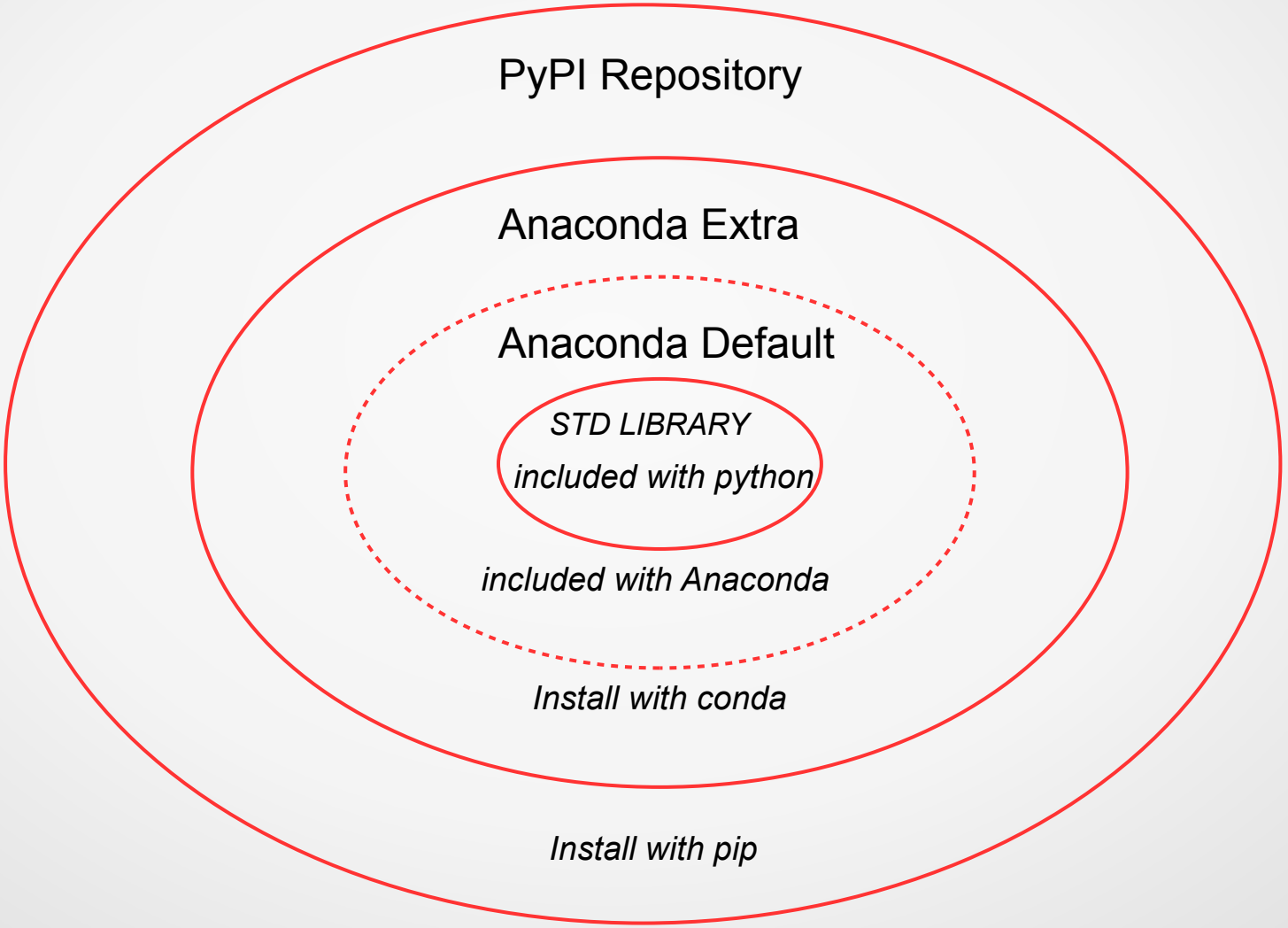
Python Modules (using Anaconda)



What Can Python Do?

- Data science
  - Data visualization
- Web apps and APIs
- Cloud apps
- Data mining/web scraping
- Desktop GUI apps
- Sys Adm (Windows, Mac, Linux)
- Scientific/Engineering apps

Advantages of Python

- Easy to learn
- Readable
- Multi-paradigm
- Modular
- Exceptions
- Large Standard library
- Many third-party modules (science, web, admin, ...)
- Fun!

Disadvantages of Python

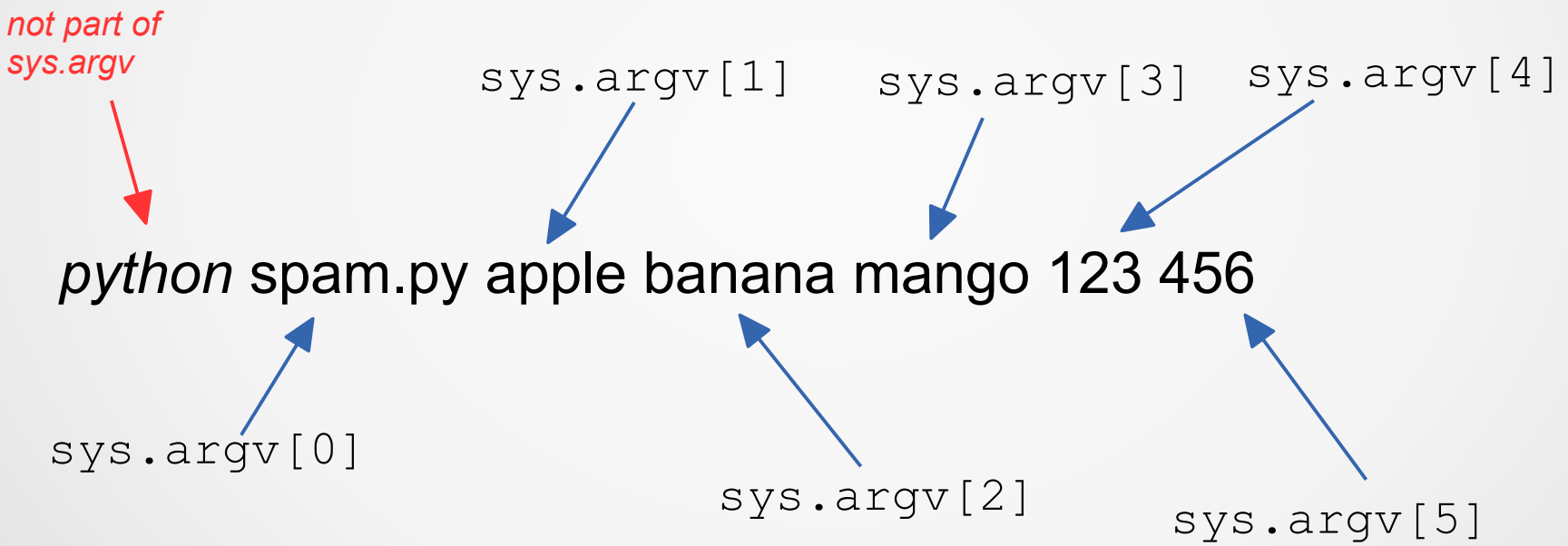
Python Evolution

String literals

- Single-delimited (AKA single-quoted)
  - `'spam\n'` `"spam\n"`
- Triple-delimited (AKA triple-quoted)
  - `'''spam\n'''` `"""spam\n"""`
- Raw
  - `r'spam\n'`

```
"Guido's the BDFL"

"""Guido's the "BDFL" of Python"""
```

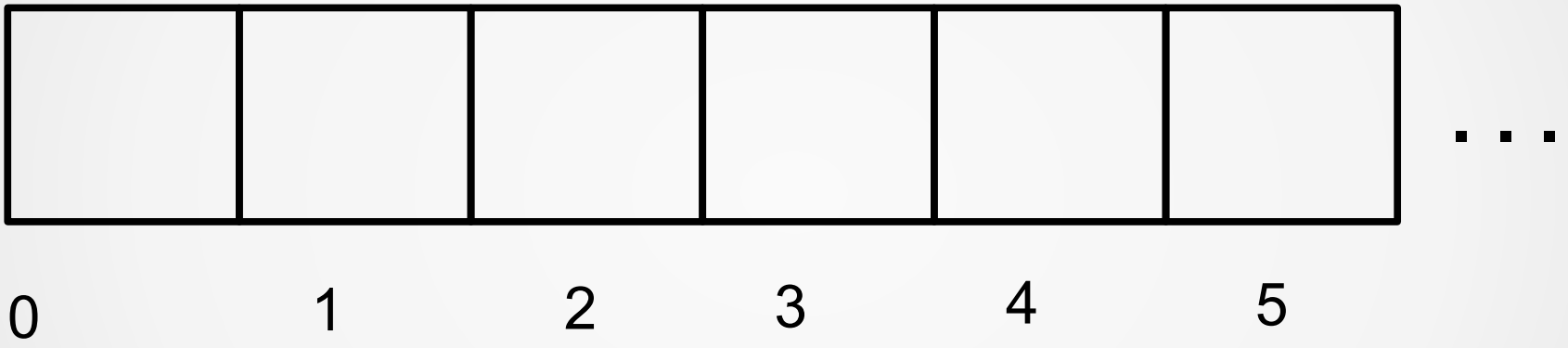
Command Line Parameters



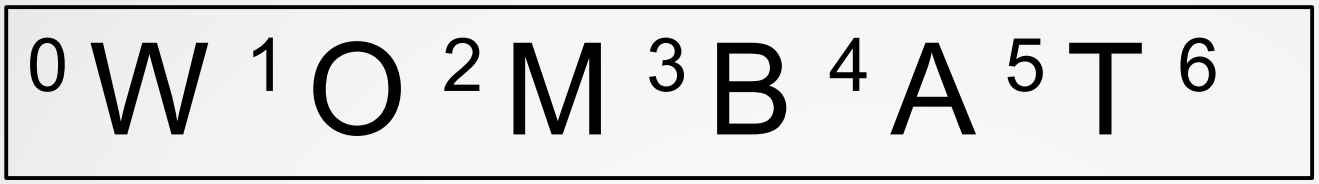
Indenting blocks

```
Block statement:
    Statement
    Statement
    Nested Block Statement:
        Statement
        Statement
    Statement
    Statement

Statement
```

Sequences



Slices




```
s = "WOMBAT"

s[0:3]       first 3 characters  "WOM"
s[:3]        same, using default start of 0 "WOM"
s[1:4]       s[1] through s[3] "OMB"
s[3:6]       s[3] through end  "BAT"
s[3:len(s)]  s[3] through end  "BAT"
s[3:]        s[3] through end, using default end "BAT"
```

Lists vs Tuples

Lists

- Dynamic Array
- Mutable/unhashable
- Position doesn't matter
- Best use: looping
- Think "ARRAY"

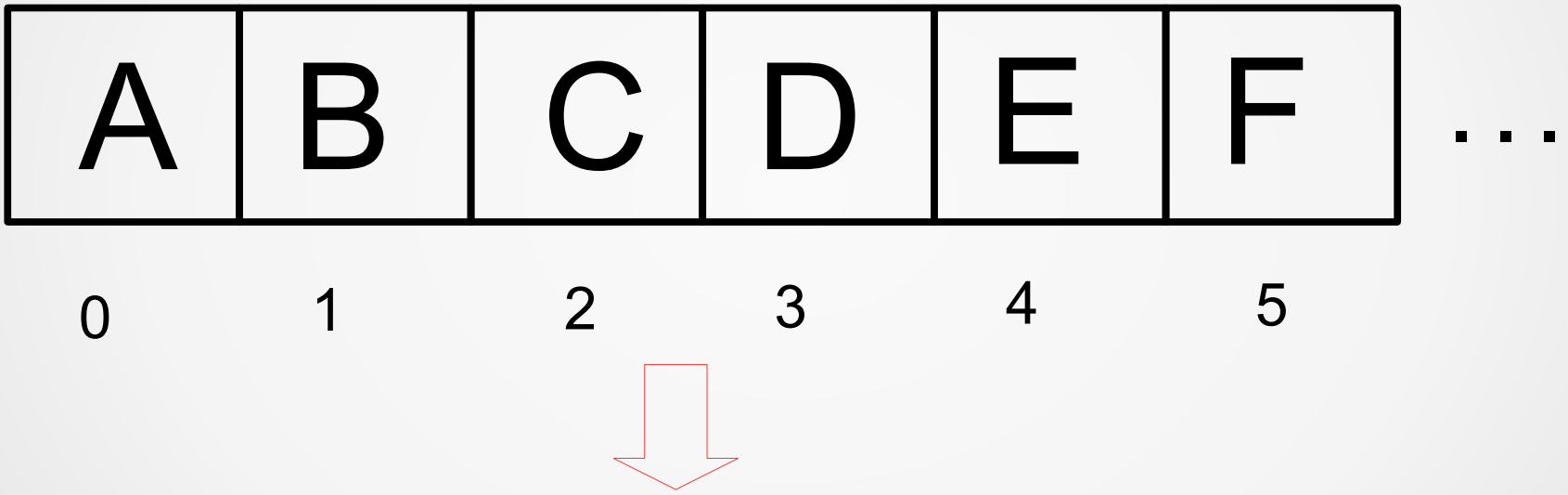
Tuples

- Collection of related fields
- Immutable/hashable
- Position matters
- Best use: unpacking
- Think "STRUCT" or "RECORD"

Myth #1: tuples are just read-only lists

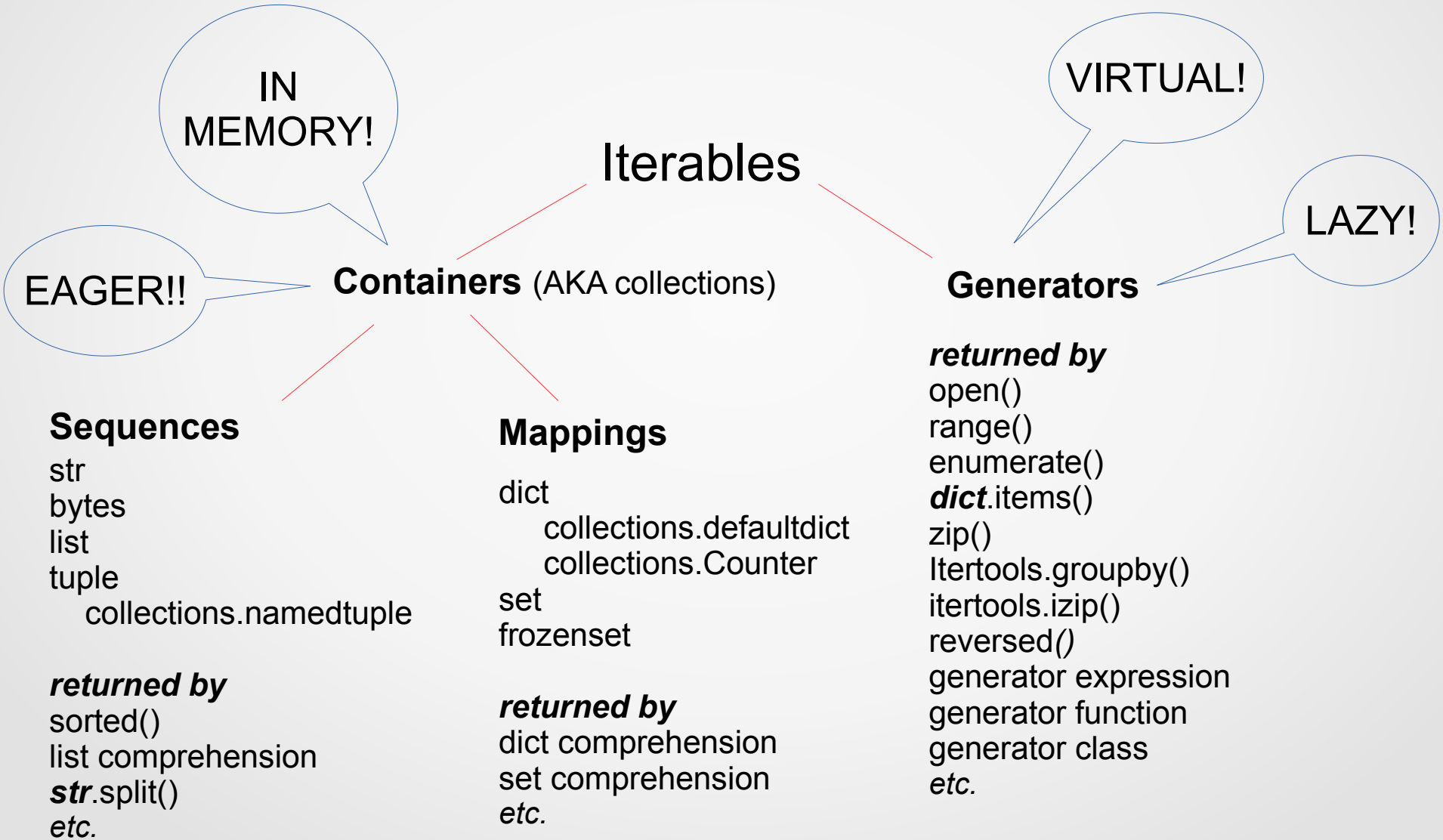
Fact #1: tuples are faster than lists (maybe only slightly)
Fact #2: tuples use less memory than lists

enumerate()

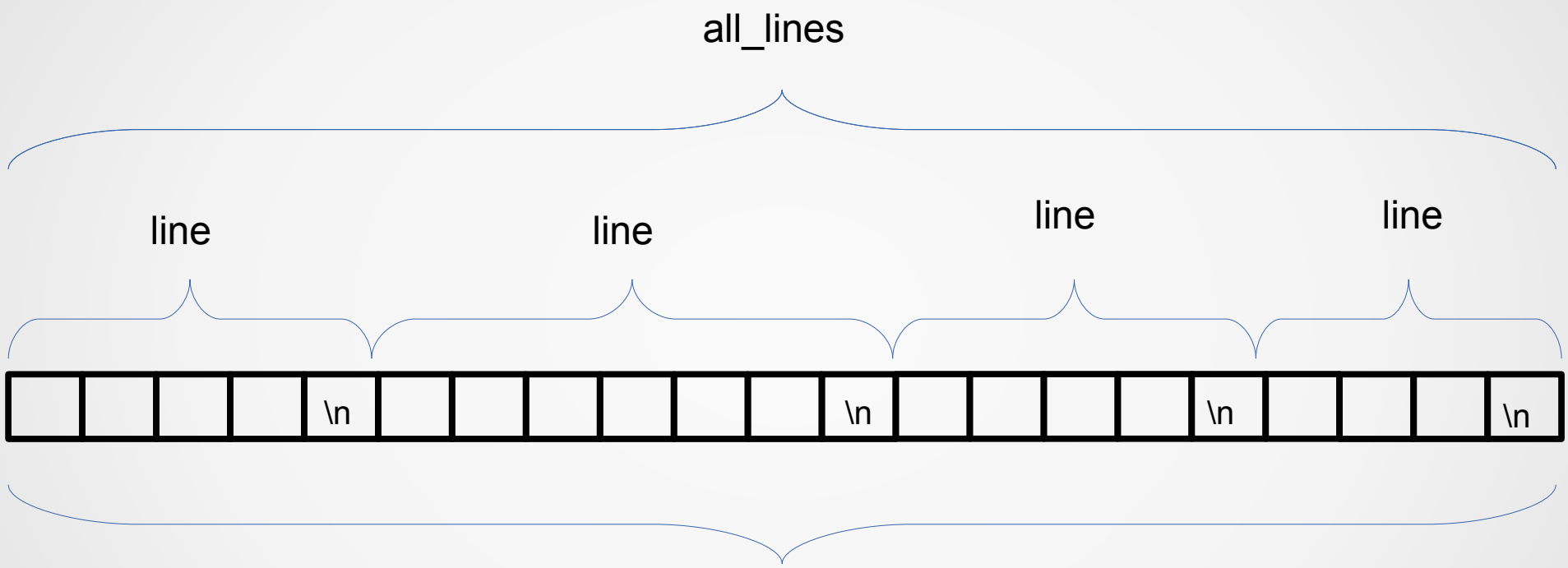


(0, A), (1, B), (2, C), (3, D), (4, E), (5, F)...

Iterables


IN MEMORY!
Iterables
VIRTUAL!
LAZY!
EAGER!!
Containers (AKA collections)
Generators
Sequences
str
bytes
list
tuple
collections.namedtuple
returned by
sorted()
list comprehension
str.split()
etc.
Mappings
dict
collections.defaultdict
collections.Counter
set
frozenset
returned by
dict comprehension
set comprehension
etc.
returned by
open()
range()
enumerate()
dict.items()
zip()
Itertools.groupby()
itertools.izip()
reversed()
generator expression
generator function
generator class
etc.

Reading text files



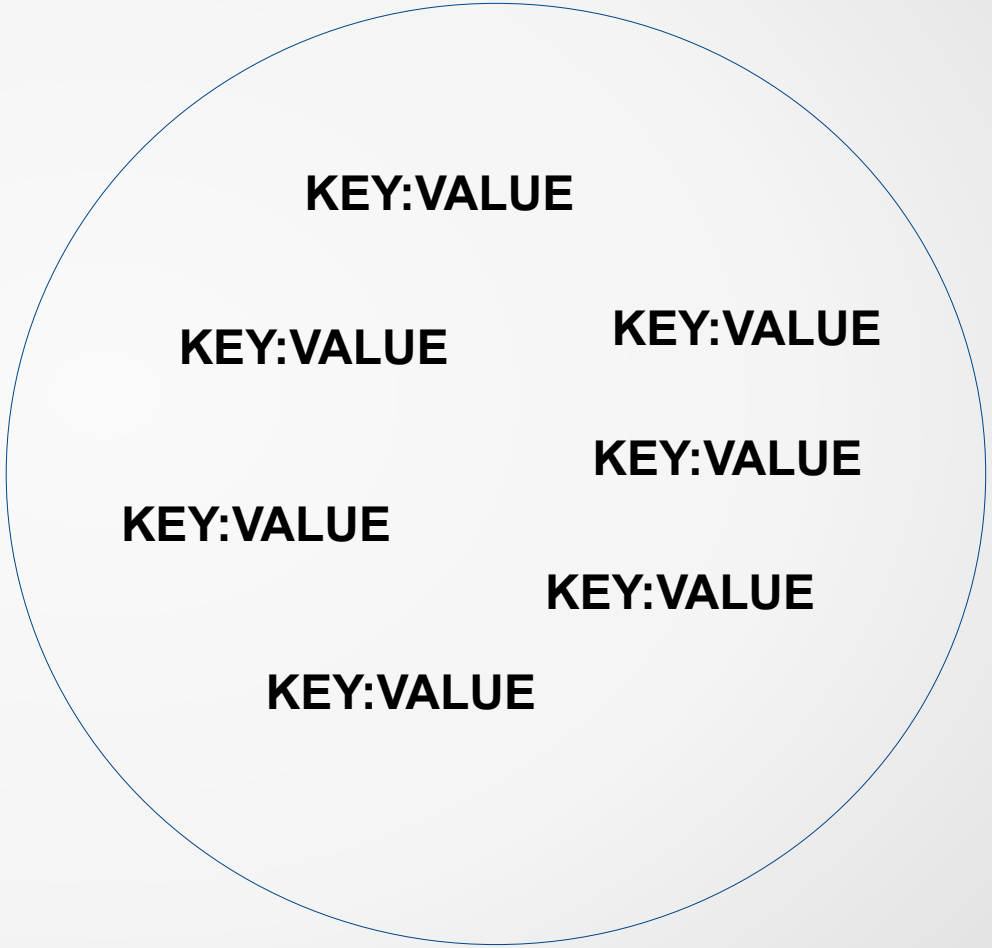

```
for line in FILE:
    pass

contents = FILE.read()

all_lines = FILE.readlines()
```

What do these words mean?

- formication
- ramiferous

Dictionary

- Key/value pairs
- Keys are unique
- Keys stored in insertion order (3.6+)
- Keys unordered (< 3.6)
- Use .items() to loop through k/v pairs
- Keys must be immutable (aka hashable)



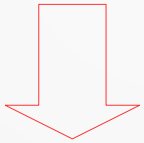
dict.items()

A	B	C	D	E	F
100	200	300	400	500	600

keys

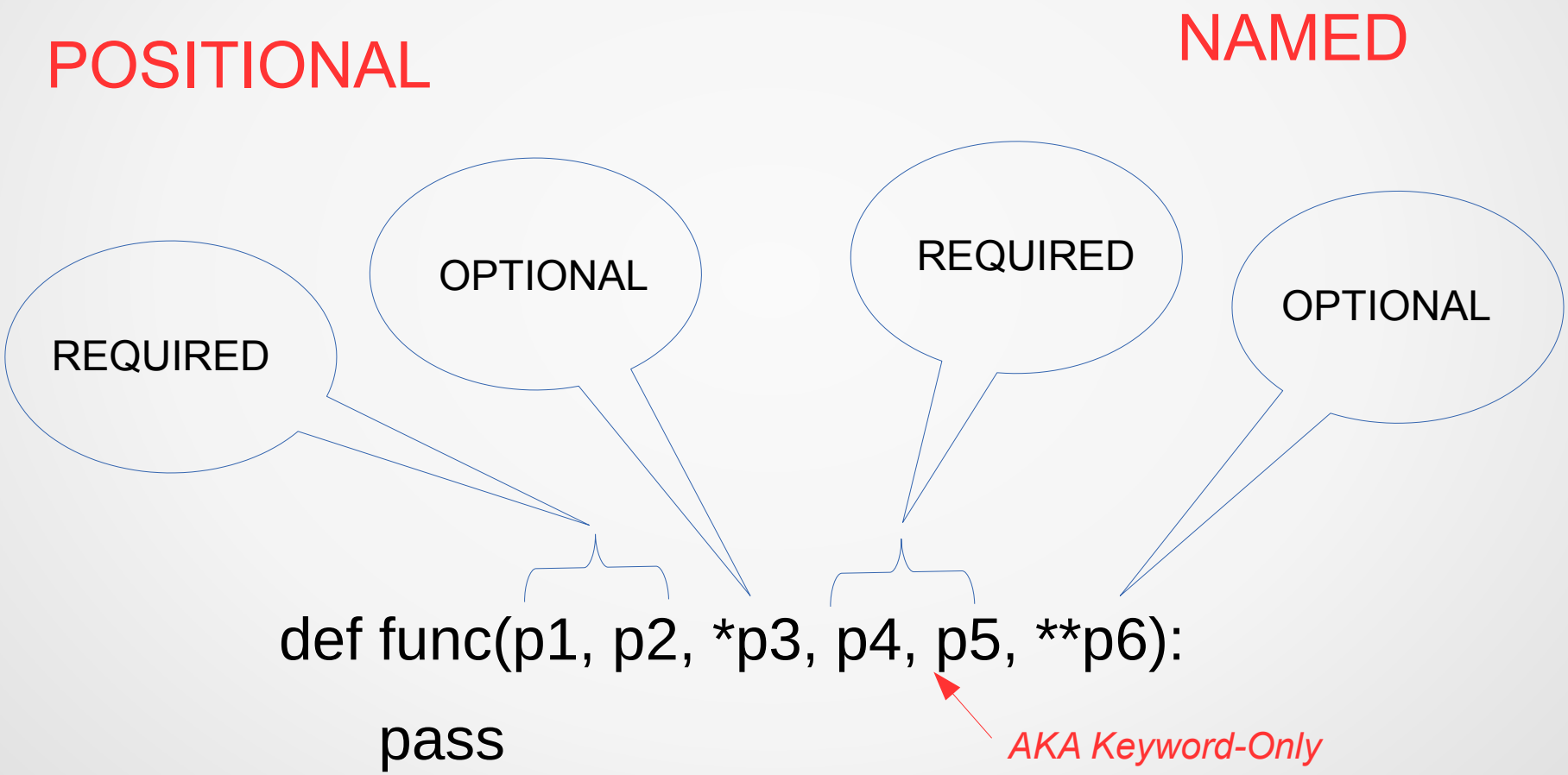
. . .

values

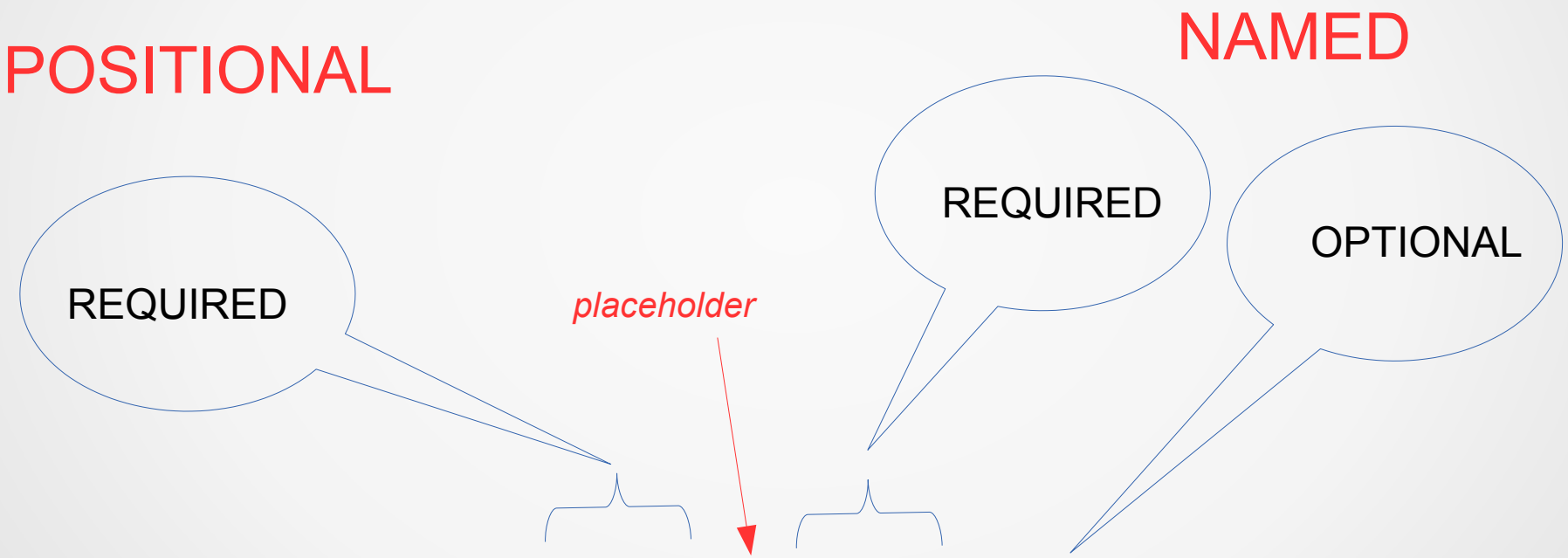


(A, 100), (B, 200), (C, 300), (D, 400), (E, 500), (F, 600) ...

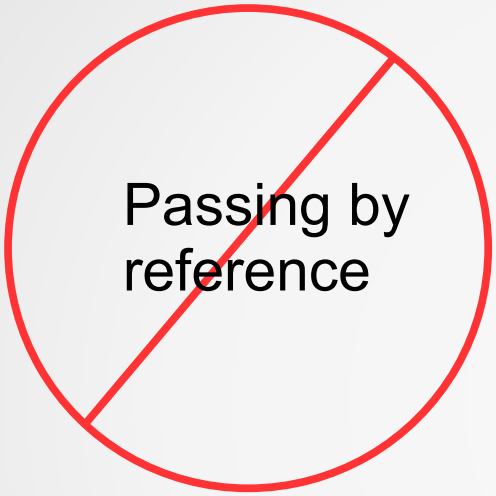
Function parameters

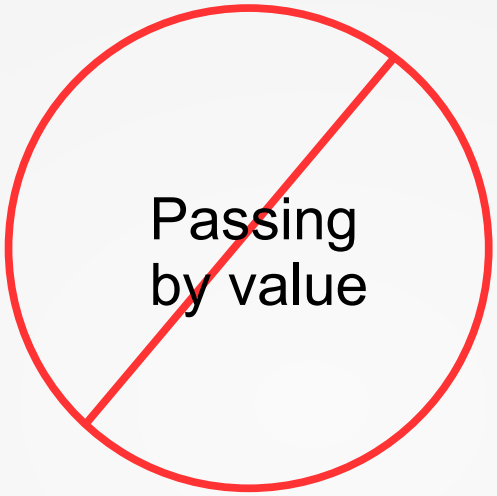


Function parameters, cont"d



Parameter passing



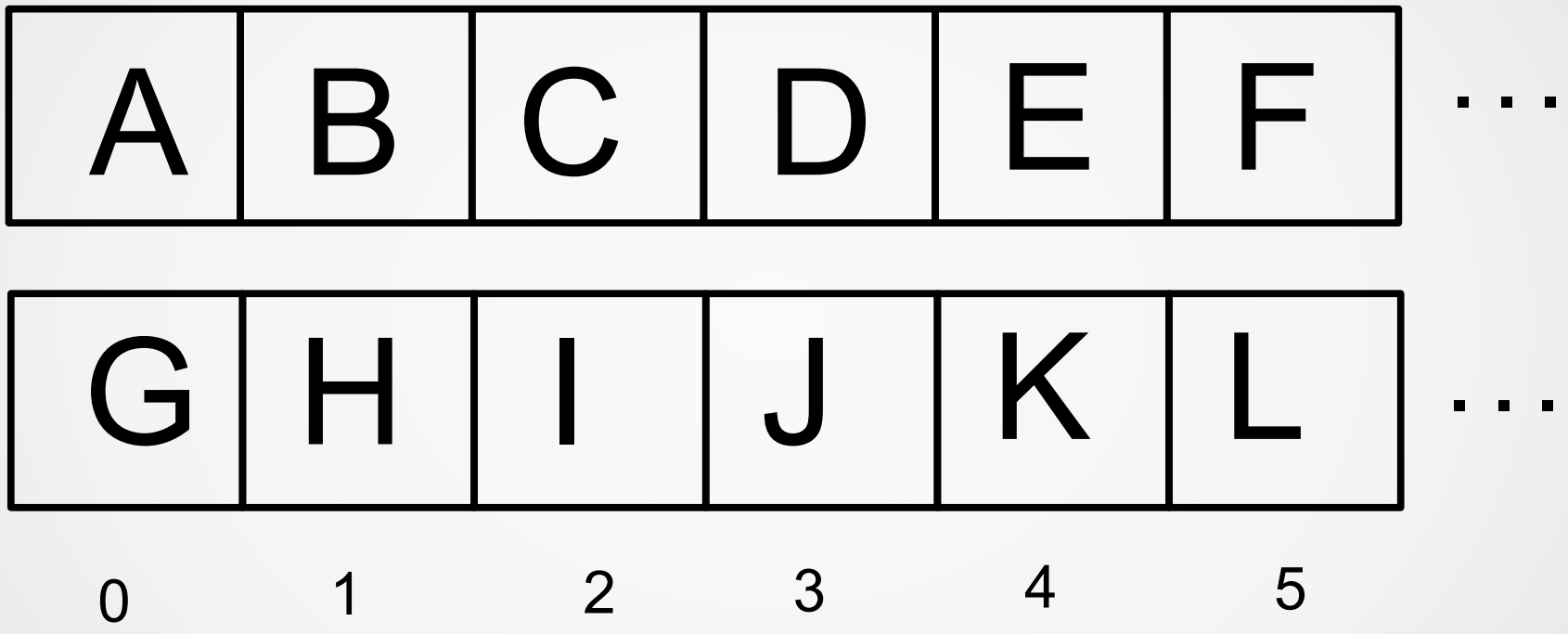


Passing by
sharing

- Read-only reference is passed
- Mutables may be changed via reference
- Immutables may not be changed

```
def spam(x, y):
    x = 5
    y.append("ham")

foo = 17
bar = ["toast", "jam"]

spam(foo, bar)
```

zip()



(A, G), (B,H), (C, I), (D, J), (E, K), (F, L)...

Sorting

- Numbers

  n, n, n, …

- Strings

  "$C_1C_2C_3$", "$C_1C_2C_3$", "$C_1C_2C_3$", …

- Nested iterables

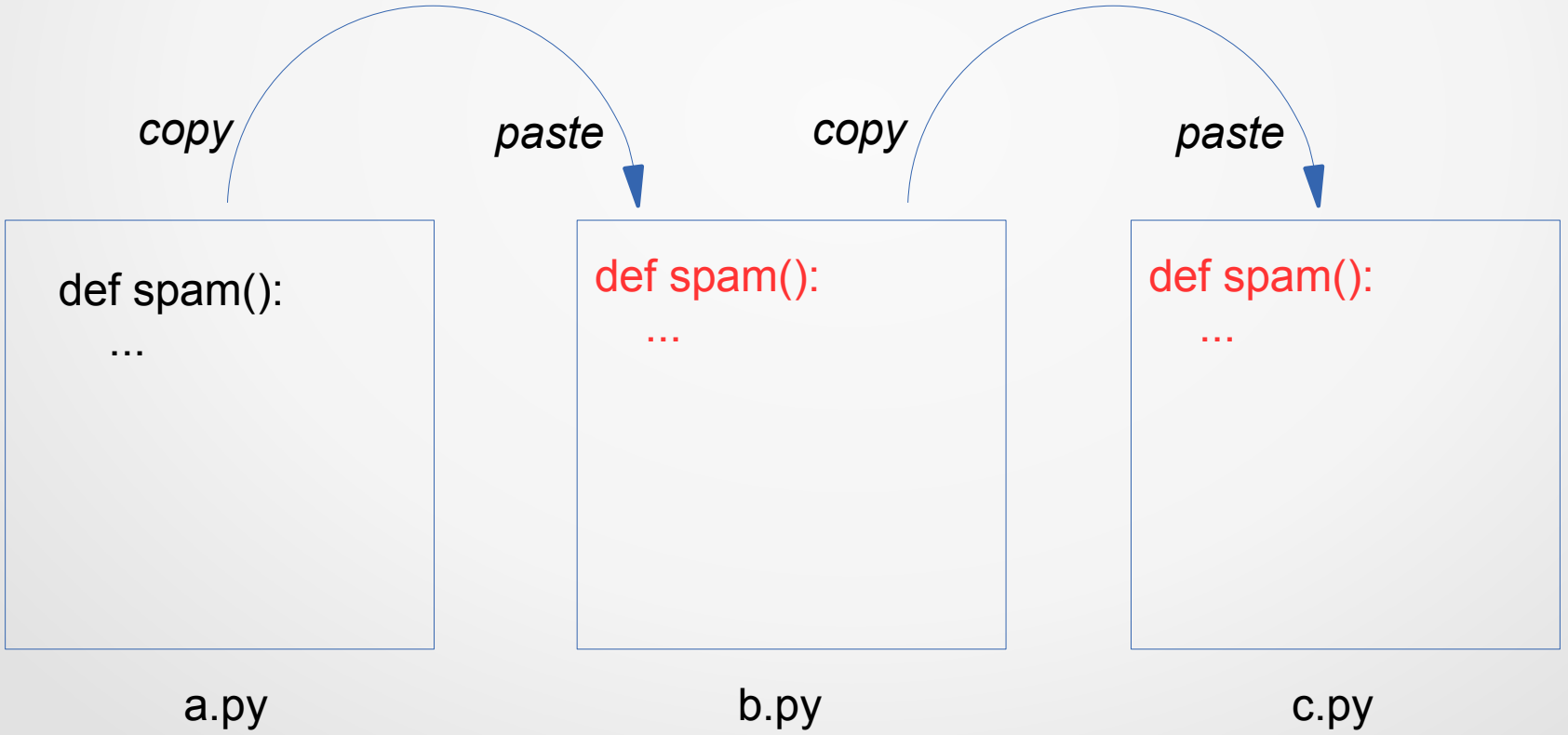
  [O_1, O_2, O_3], [O_1, O_2, O_3], [O_1, O_2, O_3], …

  - ***dict***.items() *special case of nested iterables*

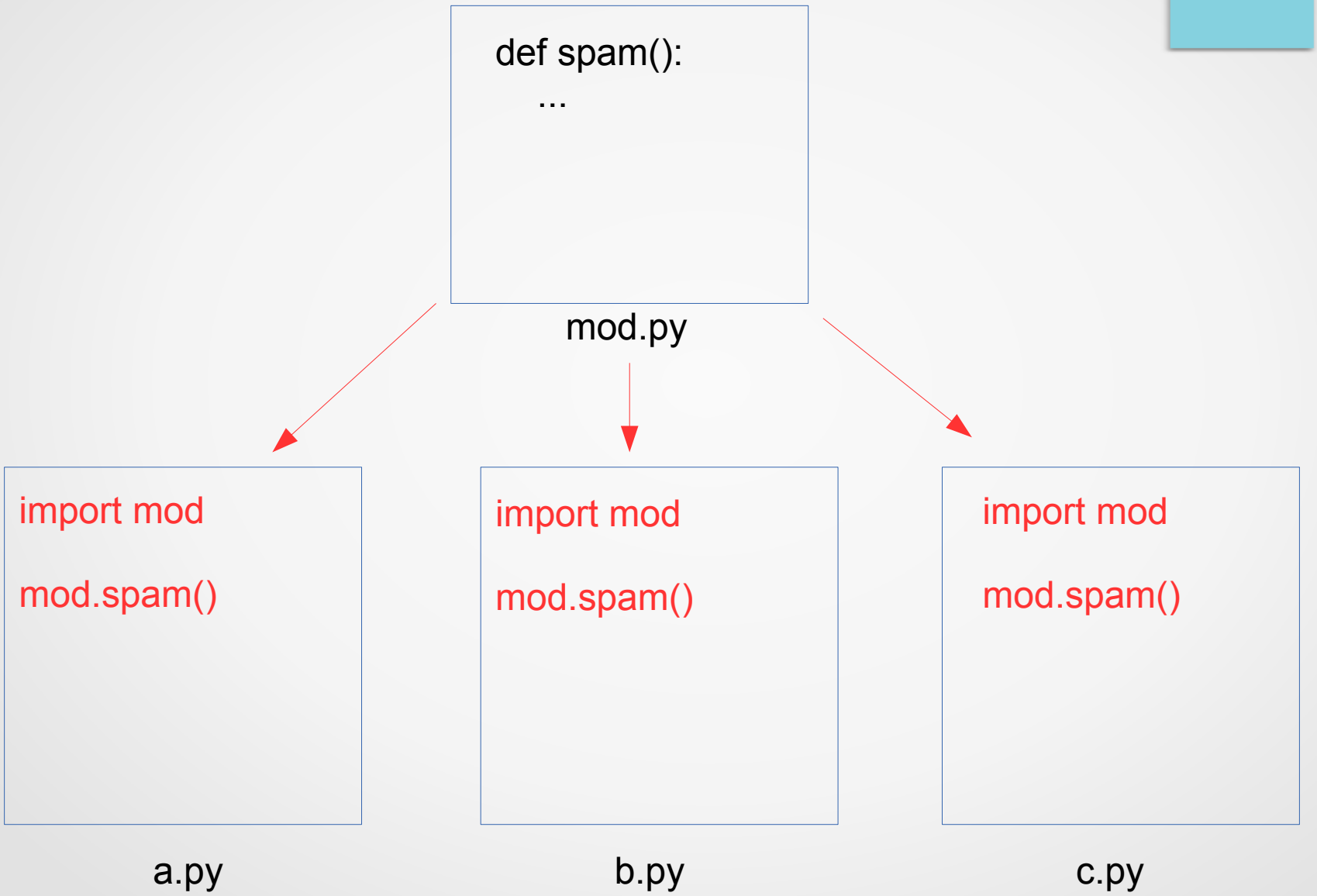
    (key, value), (key, value), (key, value), ...

Copying and pasting functions

DON'T DO THIS!!



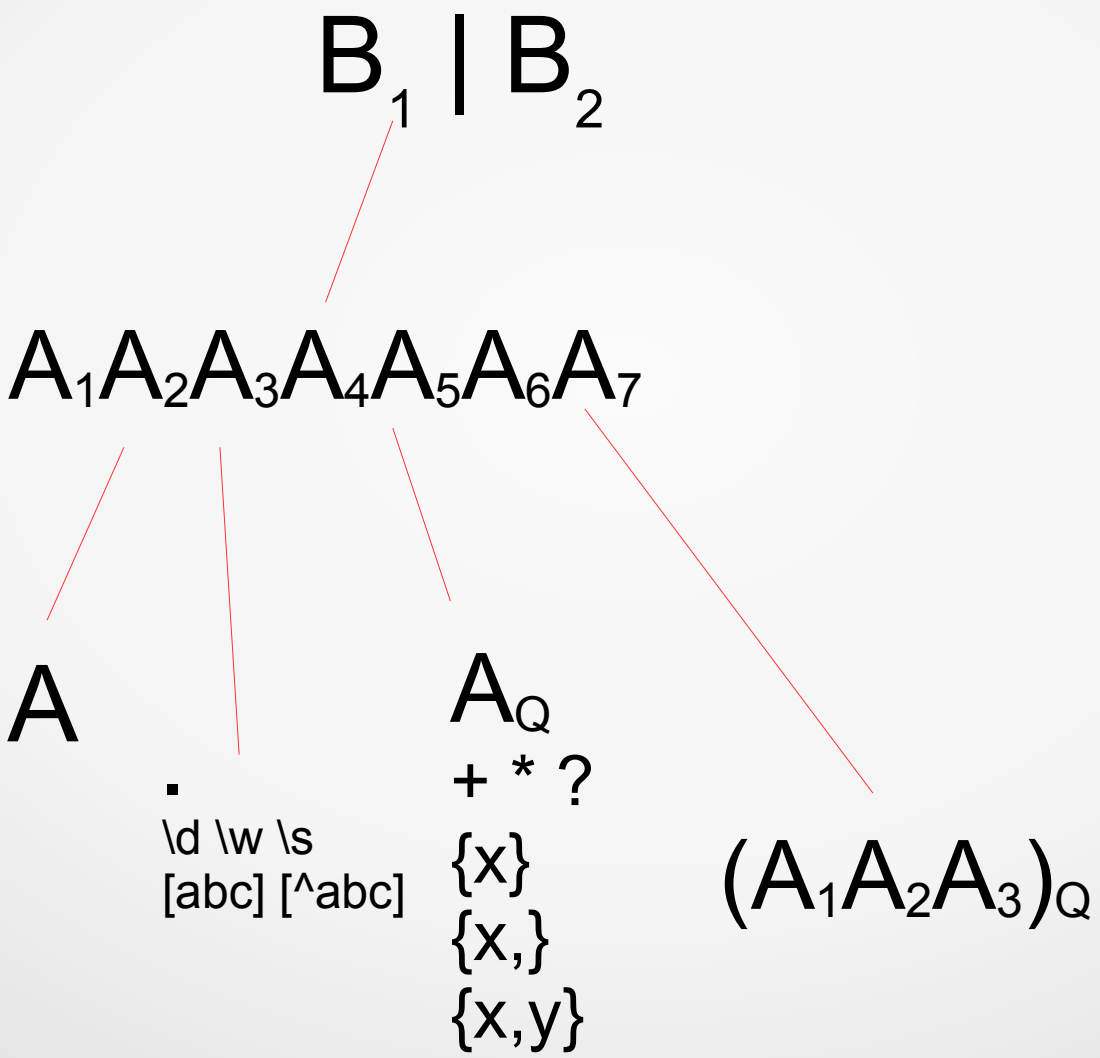
Using a module



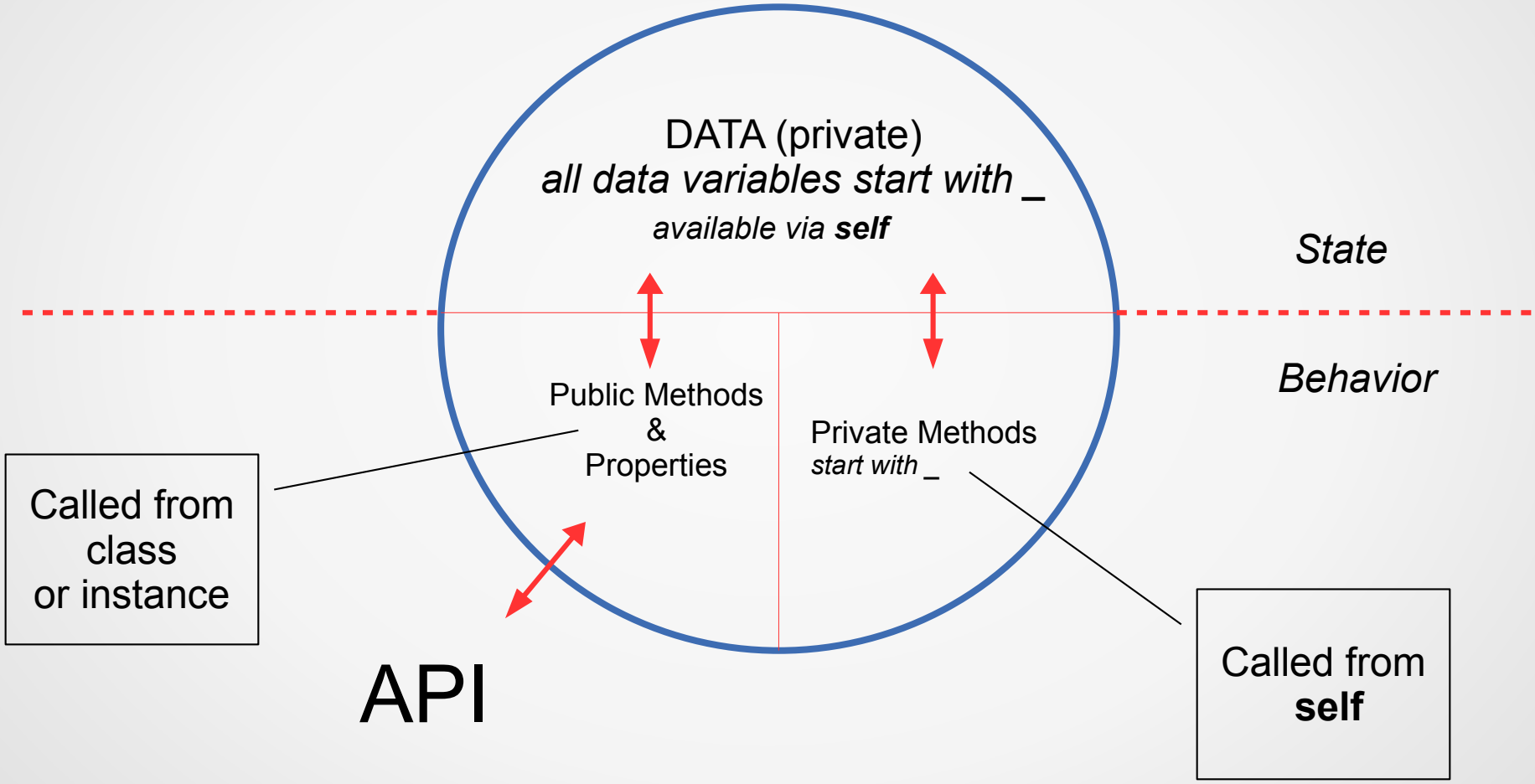
Regular expression tasks

- Search (is the match in the text?)
- Retrieve (get the matching text)
- Replace (substitute new text for match)
- Split (get what *didn't* match)

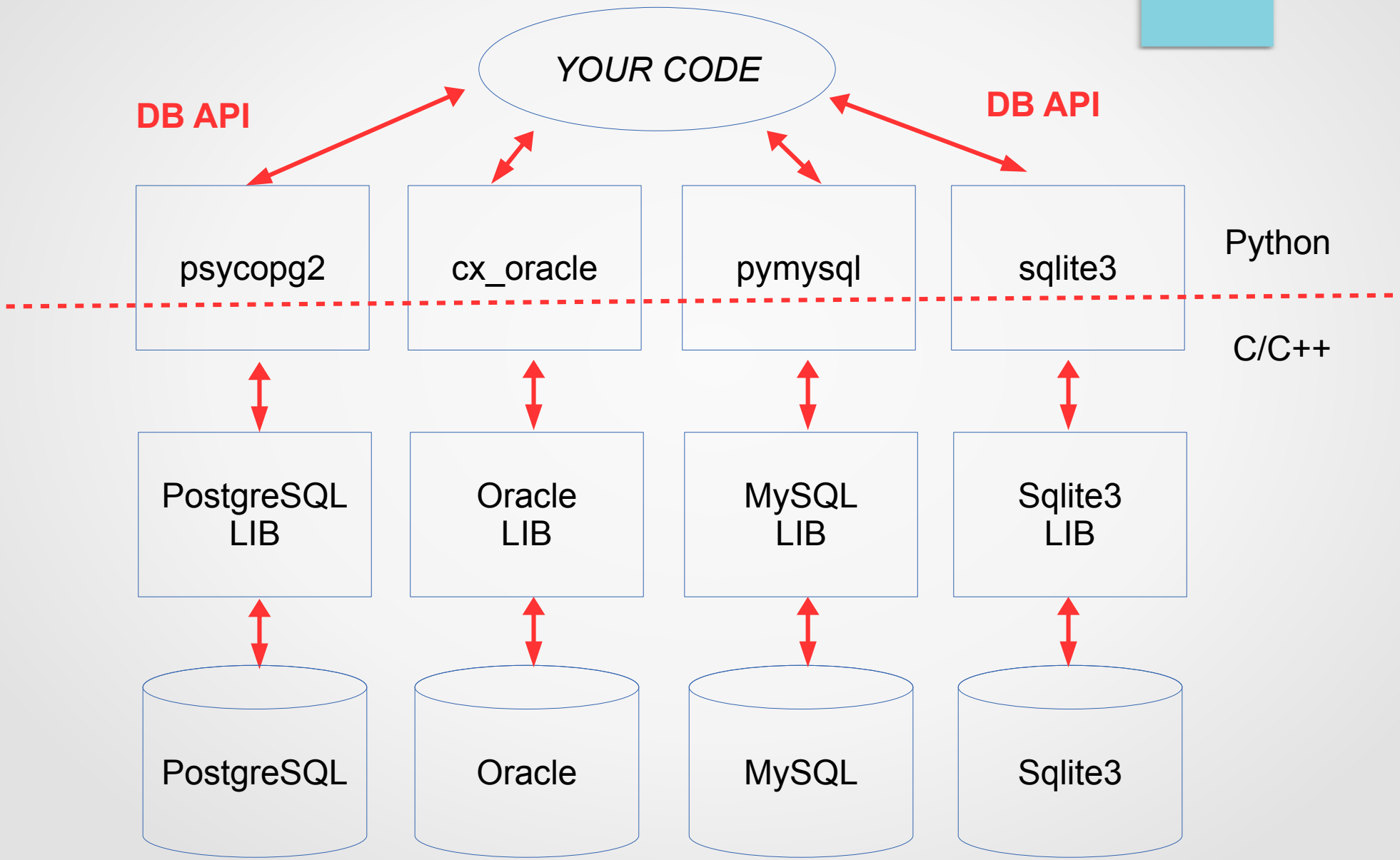
Regular Expressions



A Python Class



Python DB architecture



DB API

- conn = *package*.connect(*server, db, user, password, etc.*)
- cursor = conn.cursor()
- num_lines = cursor.execute(*query*)
- num_lines = cursor.execute(*query-with-placeholders, param-iterable)*)
- all_rows = cursor.fetchall()
- some_rows = cursor.fetchmany(n)
- one_row = cursor.fetchone()
- conn.commit()
- conn.rollback()

How a *for* loop really works

values = ["a", "b", "c"]

for loop:

```
for value in values:
    print(value)
```

while loop:

```
it = iter(values)
while True:
    try:
        value = next(it)
    except StopIterationError:
        break
```

SqlAlchemy ORM

DBMS Table

```
create table person (
    id int autoincrement,
    firstname varchar(30),
    lastname varchar(30),
    age int,
)
```

Python class

```
class person(base):
  id = Column(
      Integer,
      primary_key=True
  )
  last_name = Column(String(50))
  first_name = Column(String(50))
  age = Column(Integer)
```

ElementTree

presidents.xml

```
<presidents>
    <president term="1">
        <lastname>Washington</lastname>
        <firstname>George</firstname>
    </president>
    <president term="2">
        <lastname>John</lastname>
        <firstname>Adams</firstname>
    </president>
</presidents>
```

ElementTree

```
Element
      tag="presidents"
  Element {"term": "1" }
    tag="president"
      Element
        tag="lastname"
        text="Washington"
      Element
        tag="firstname"
        text="George"
  Element {"term": "2" }
    tag="president"
      Element
        tag="lastname"
        text="Adams"
      Element
        tag="firstname"
        text="John"
```

Good sources of Python books

- http://www.packtpub.com
- http://www.oreilly.com

Accessing Excel from Python

- pandas.read_excel()
- openpyxl
- win32com (requires Excel to be running)
- use CSV/TSV
- xlrd, xlwt, xlutil