Python Programming Boot Camp

TTPS4814

IDE Features

- Autocomplete
- Autoindent
- Syntax checking/highlighting
- Debugging
- Integration with source code control (e.g. git)
- Navigation
- Smart search-and-replace

IDE Features

- Project management
- Code snippets (AKA macros)
- File templates
- Variable explorer
- Python console
- Interpreter configuration (including installing modules)
- Unit testing tools

Standard library

- 300+ modules
- Always available

Creating Variables

```
x = 5
```

Creating Variables

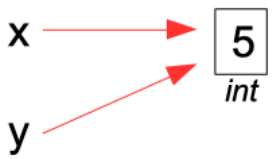
```
x = 5
```

Creating Variables

```
x = 5
y = x
```

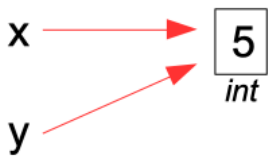
Creating Variables

```
x = 5
y = x
```



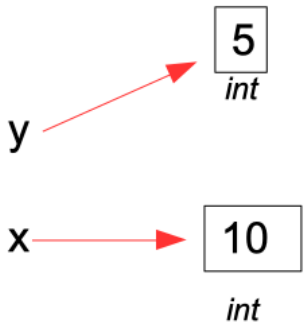
Creating Variables

```
x = 5
y = x
x = 10
```



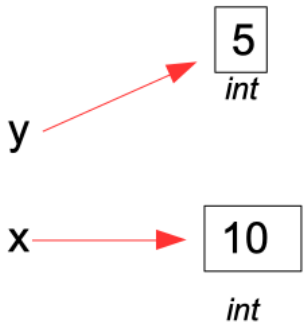
Creating Variables

```
x = 5
y = x
x = 10
```



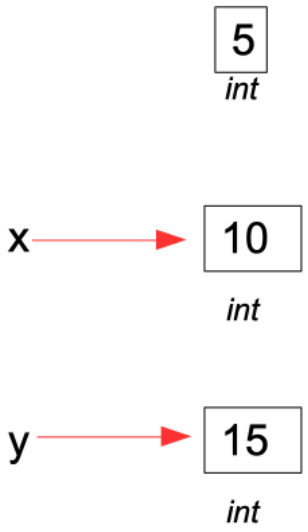
Creating Variables

```
x = 5
y = x
x = 10
y = 15
```



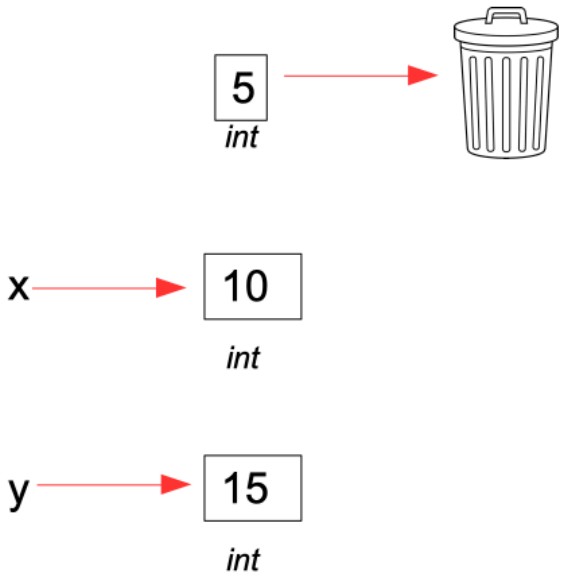
Creating Variables

```
x = 5
y = x
x = 10
y = 15
```



Creating Variables

```
x = 5
y = x
x = 10
y = 15
```



String literals

- Three flavors
  - single-delimited
  - triple-delimited
  - raw

Single-delimited

- Use either single or double quote character

```
"spam\n"
'spam\n'

print("Guido's the bomb!")
print('Guido is the "benevolent" dictator of Python')
```

Triple-delimited

- Single or double quote character
- No need to escape quotes

```
"""spam\n"""
'''spam\n'''

query = """
    select *
    from logs
    where date > '2018-02-19'
"""

print('''Guido's the "benevolent" dictator of Python''')
```

Raw

- Does not interpret backslashes

```
r"spam\n"
r'spam\n'
```

`str()` vs `repr()`

`str()`	**`repr()`**
For humans	How to **repr**oduce object
"Informal" form	"Official" form
Info about object	Code to create object
if undefined, uses `repr()`	

f-string shortcut

Instead of

```
print(f"x = {x})
```

use

```
print(f"{x = }")
```

x is only typed once

Command line arguments

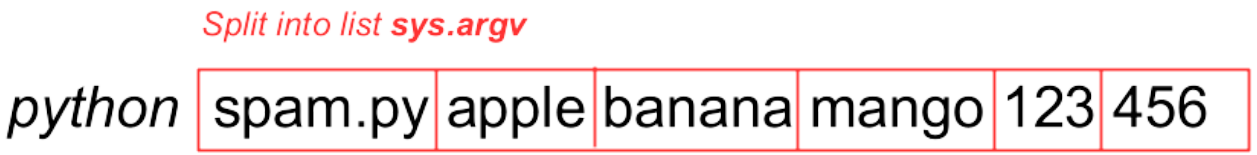
python spam.py apple banana mango 123 456

Command line arguments

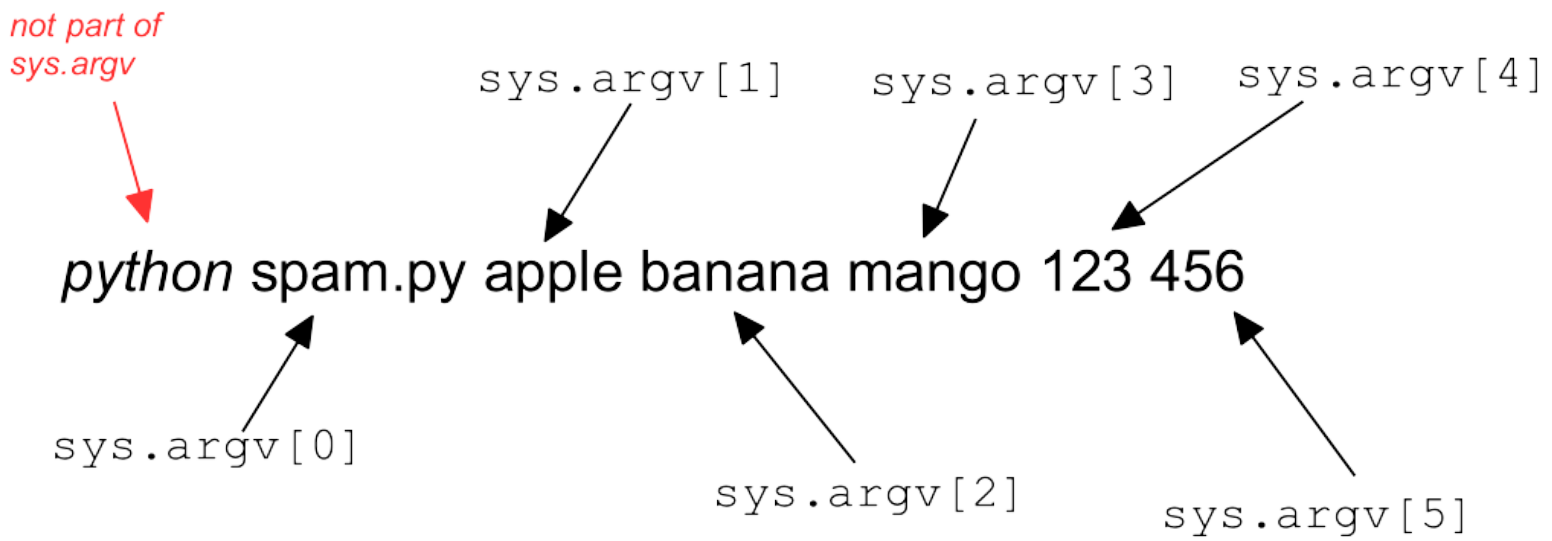
All arguments to python interpreter

python spam.py apple banana mango 123 456

Command line arguments



Command line arguments



Indenting blocks

```
value = 56
if value > 75:
    print("wombat")
    print("wallaby")
elif value > 50:
    print("kangaroo")
    print("kookaburra")
    print("koala")
    if value > 60:
        print('cane toad')
```

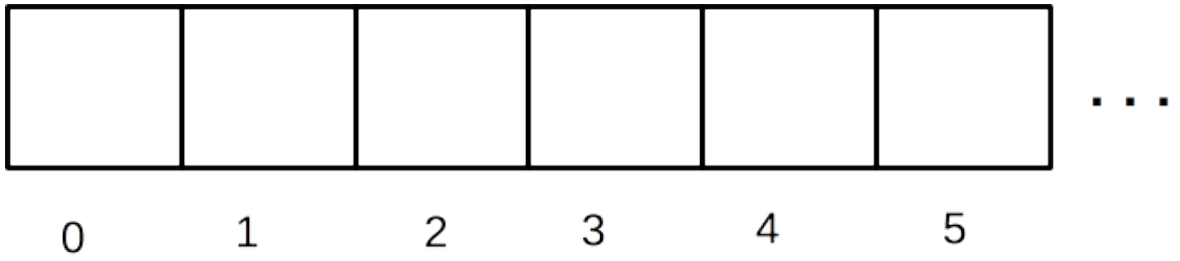
Boolean values

If X is	Boolean value of X is
Numeric, and equal to 0	`False`
Numeric, and NOT equal to 0	`True`
A collection, and len(**X**) is 0	`False`
A collection, and len(**X**) is > 0	`True`

Boolean values

If X is	Boolean value of X is
`None`	`False`
`False`	`False`
`True`	`True`
anything else	`True`

Sequences




```
colors = ['purple', 'orange', 'black']
print(colors[1])   # prints 'orange'
for color in colors:
    print(color)
```

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	Tuples
Dynamic array	Collection of related fields
Mutable/unhashable	Immutable/hashable
Position doesn’t matter	Position matters
Designed for iterating	Designed for unpacking
"ARRAY"	"STRUCT" or "RECORD"

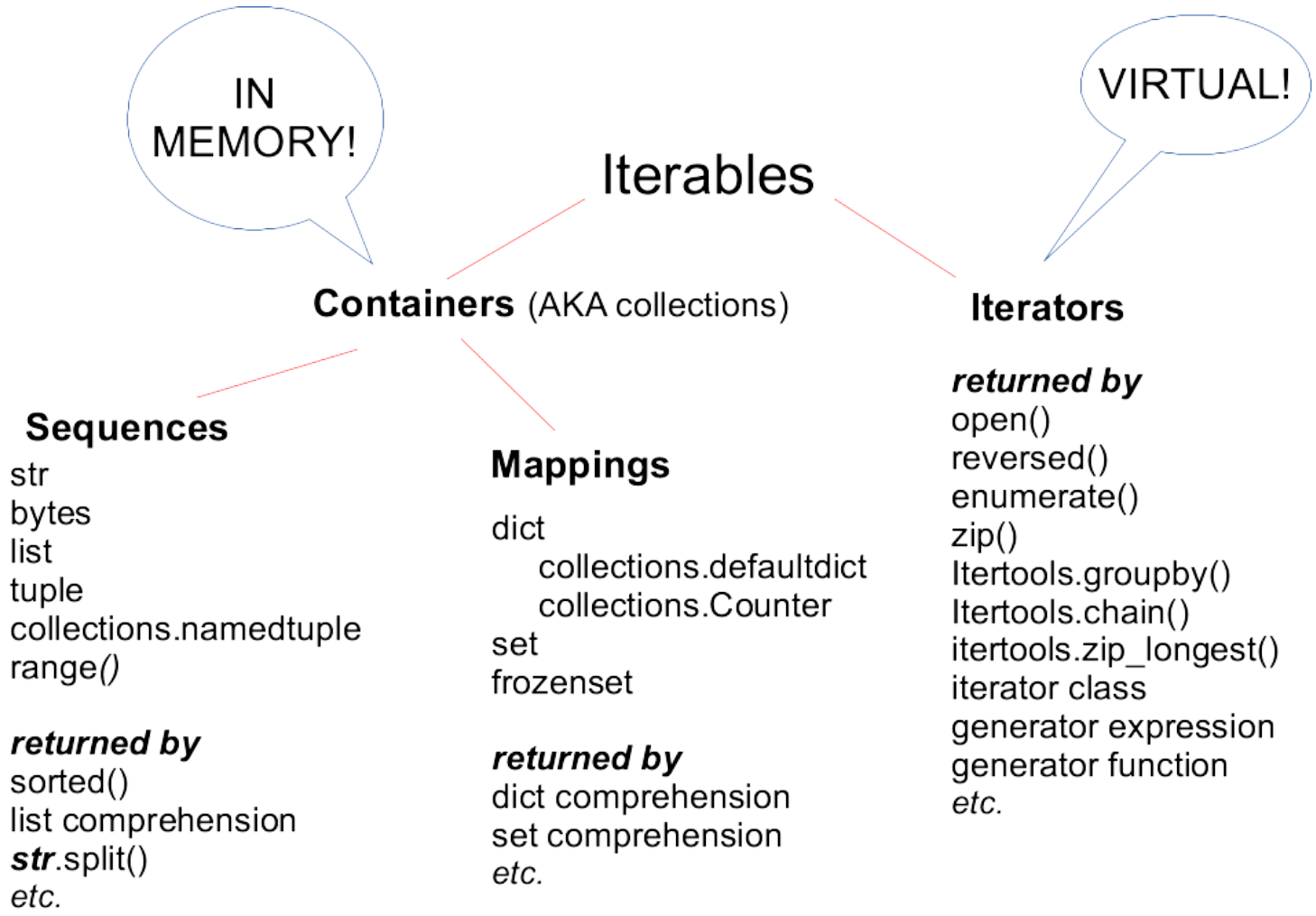
A Myth

Tuples are just read-only lists

Tuple alternatives

- Standard library
  - namedtuple
  - dataclass
- Third-party
  - attrs
  - Pydantic

Iterables


IN MEMORY!
Iterables
VIRTUAL!
Containers (AKA collections)
Iterators
Sequences
str
bytes
list
tuple
collections.namedtuple
range()
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()
reversed()
enumerate()
zip()
Itertools.groupby()
Itertools.chain()
itertools.zip_longest()
iterator class
generator expression
generator function
etc.

Containers

- All elements in memory
- Can be indexed with [ ]
- Have a length

Builtin containers

Sequences

`list`

`tuple`

`string`

`bytes`

`range`

Mapping types

`dict`

`set`

`frozenset`

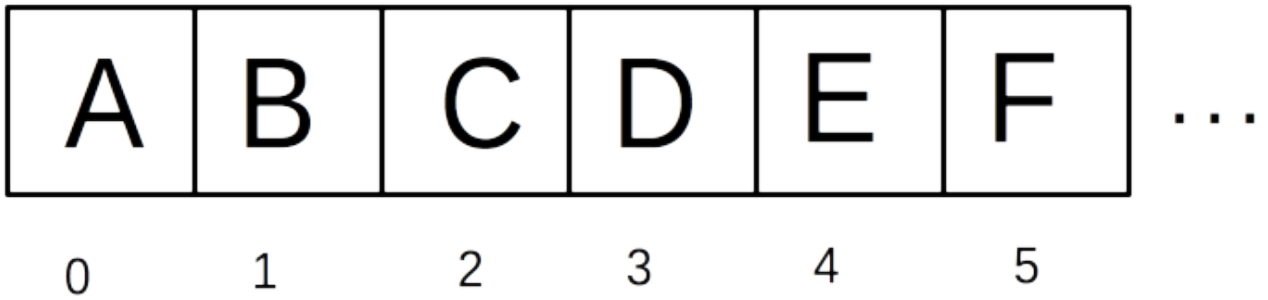
Iterators

- Virtual (no memory used for data)
- Lazy evaluation (JIT)
- Cannot be indexed with [ ]
- Do not have a length
- One-time-use

Iterators returned by

- `open()`
- `enumerate()`
- *`DICT`*`.items()`
- `zip()`
- `reversed()`
- *generator expression or function*
- *iterator class*

enumerate



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

Using `enumerate()`

```
letters = ['alpha', 'beta', 'gamma']  # or any iterable...
```

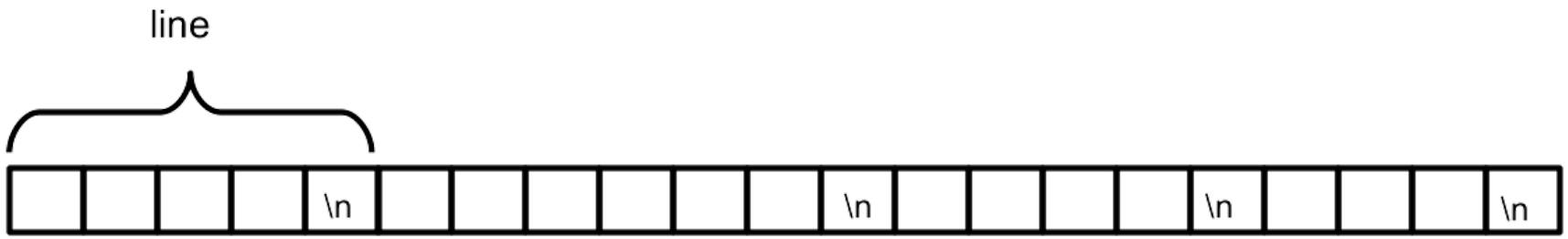
```
enumerate(letters)
(0, 'alpha'), (1, 'beta'), (2, 'gamma')
```

```
enumerate(letters, 1)
(1, 'alpha'), (2, 'beta'), (3, 'gamma')
```

Reading Text Files

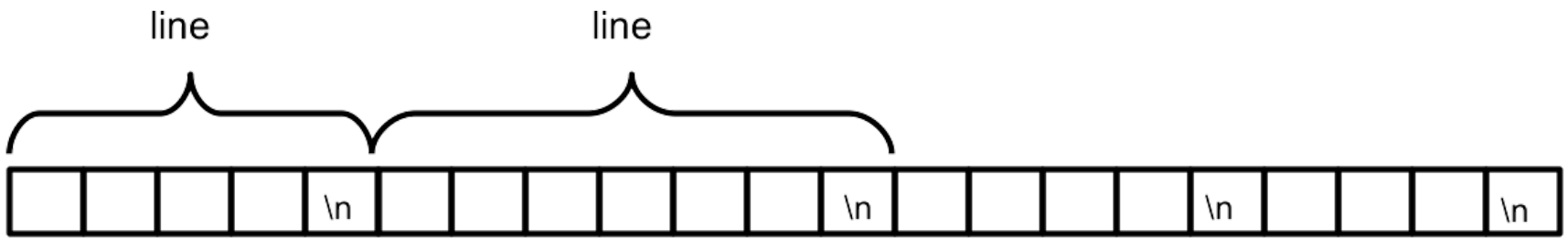

```
with open("somefile") as file_in:
```

Reading one line at a time



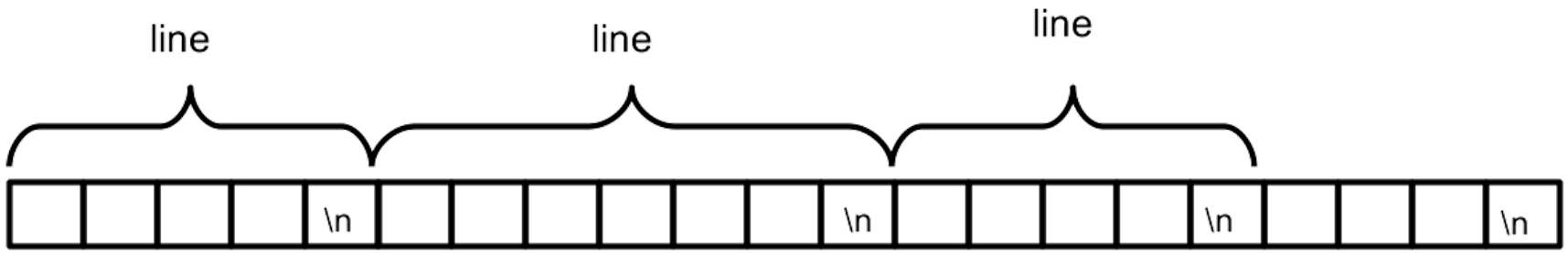

```
with open("somefile") as file_in:
    for raw_line in file_in:
        ...
```

Reading one line at a time



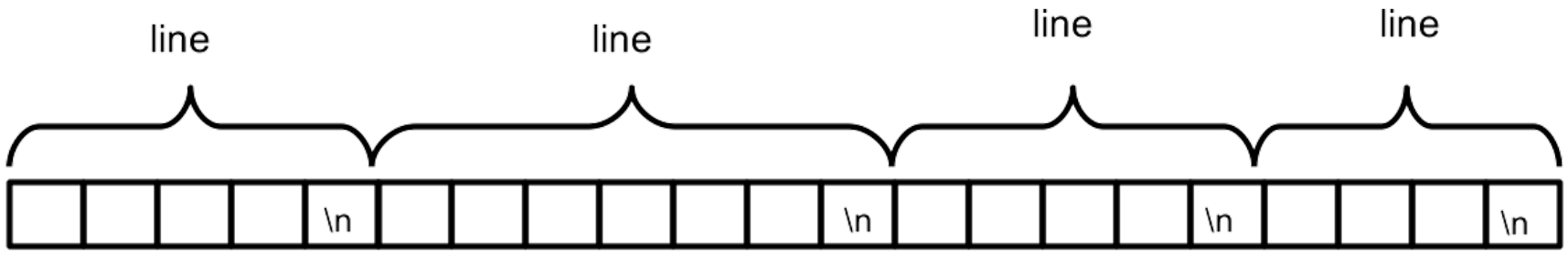

```
with open("somefile") as file_in:
    for raw_line in file_in:
        ...
```

Reading one line at a time



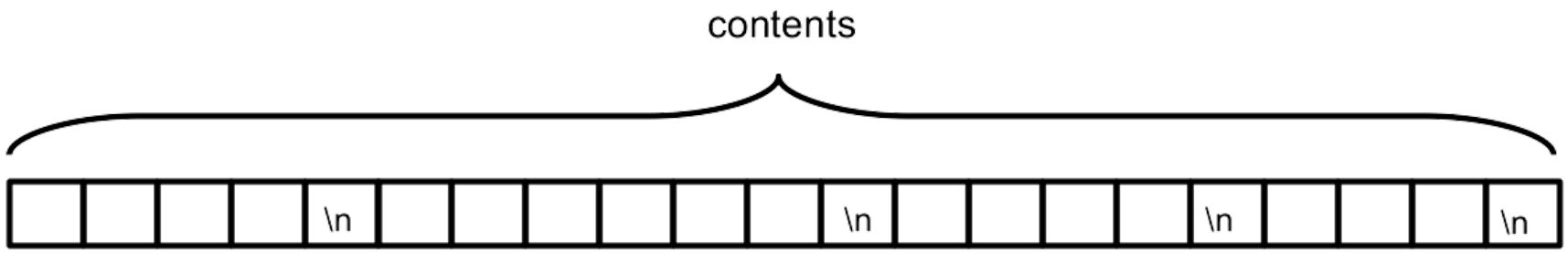

```
with open("somefile") as file_in:
    for raw_line in file_in:
        ...
```

Reading one line at a time



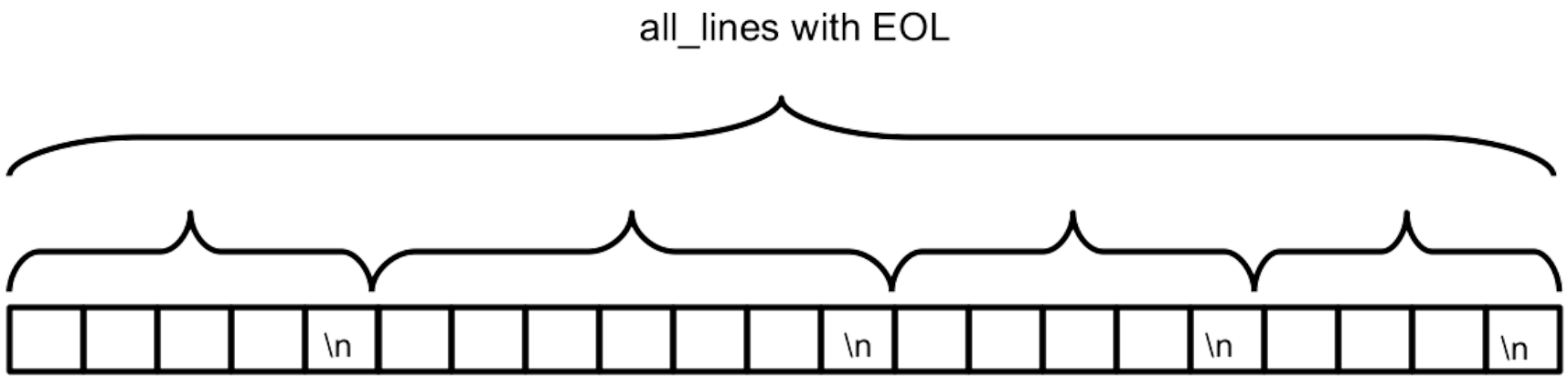

```
with open("somefile") as file_in:
    for raw_line in file_in:
        ...
```

Reading entire file into string



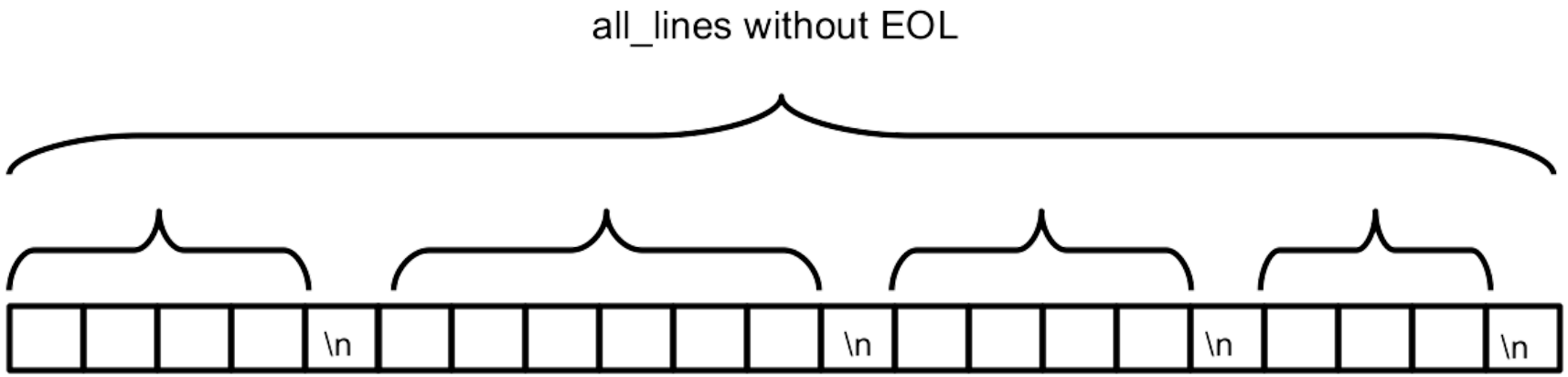

```
with open("somefile") as file_in:
    contents = file_in.read()
```

Reading file into list of strings (with EOL)




```
with open("somefile") as file_in:
    all_lines = file_in.readlines()
```

Reading file into list of strings (without EOL)




```
with open("somefile") as file_in:
    all_lines = file_in.read().splitlines()
```

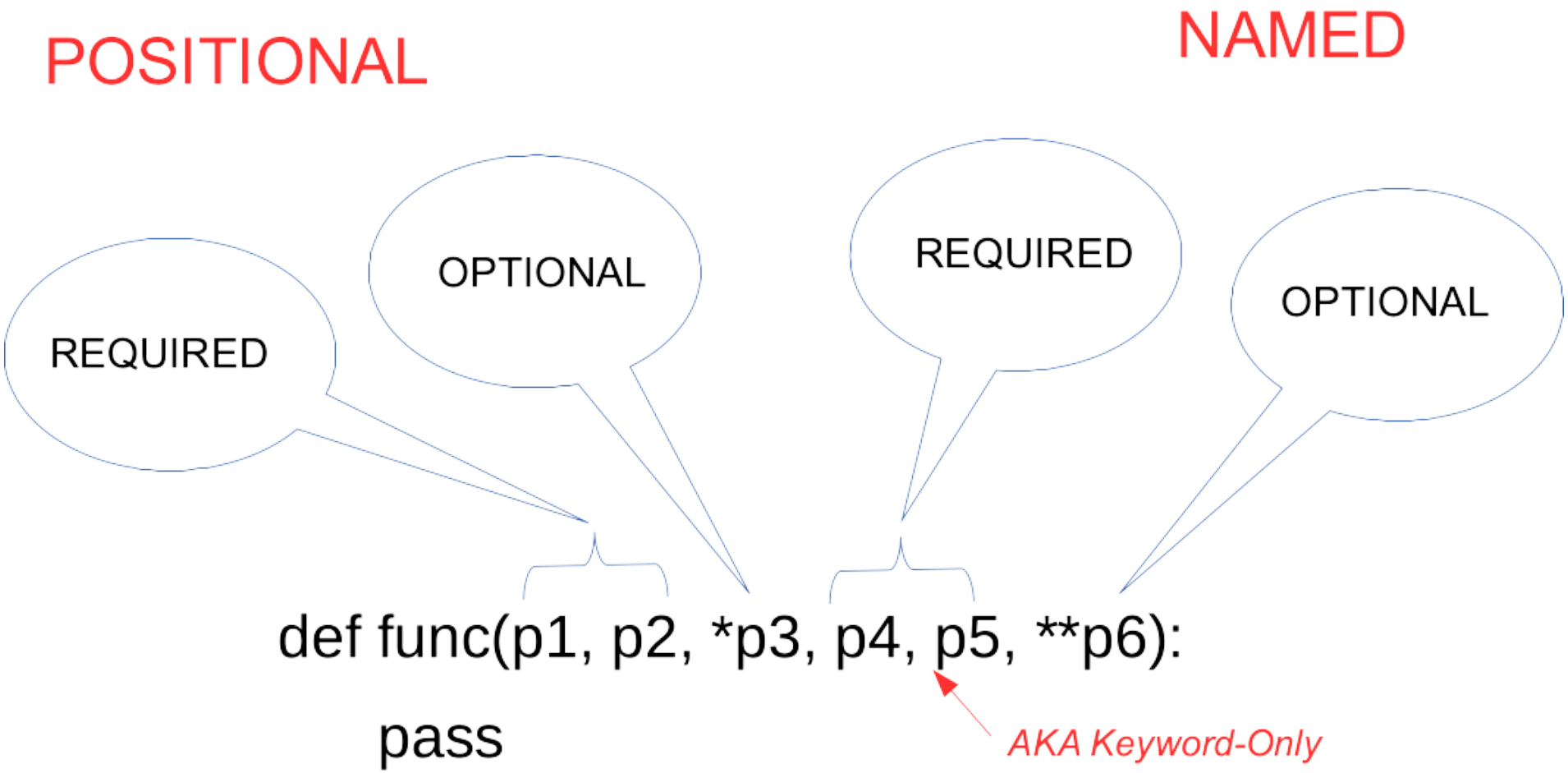
Dictionary

- Key/value pairs
- Keys must be immutable
  - str
  - int, float
  - tuple
- Keys are unique
- Keys/values stored in insertion order

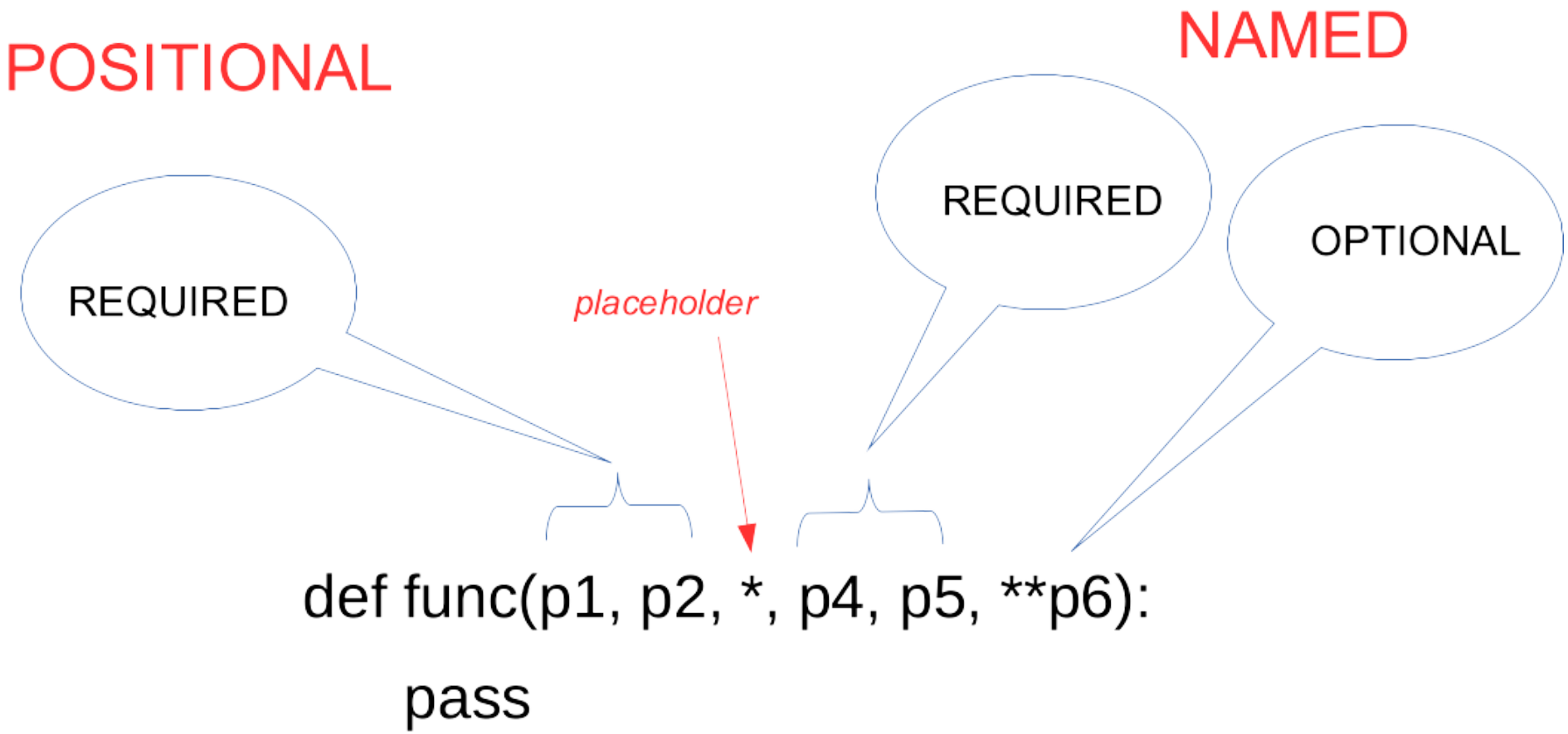
Dictionary items

```
for key, value in _DICT_.items():
    ...  # use key or value here
```

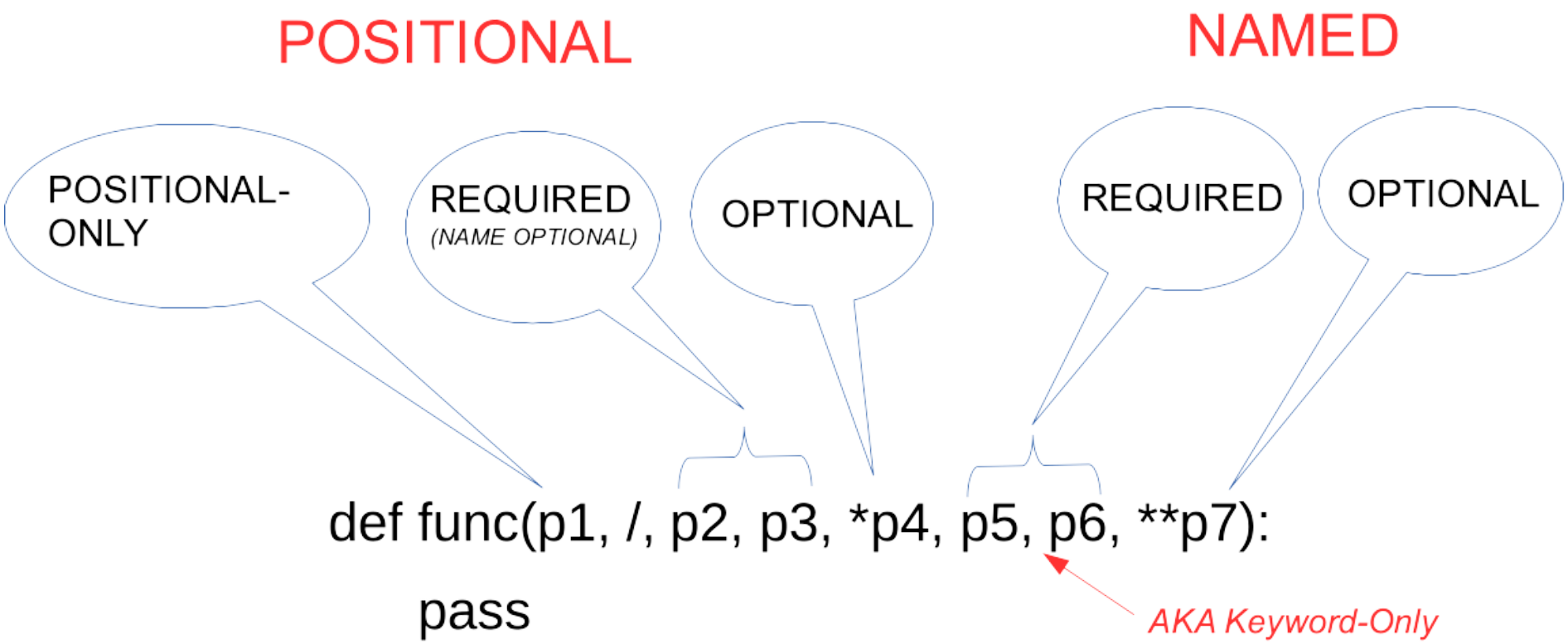
Function parameters



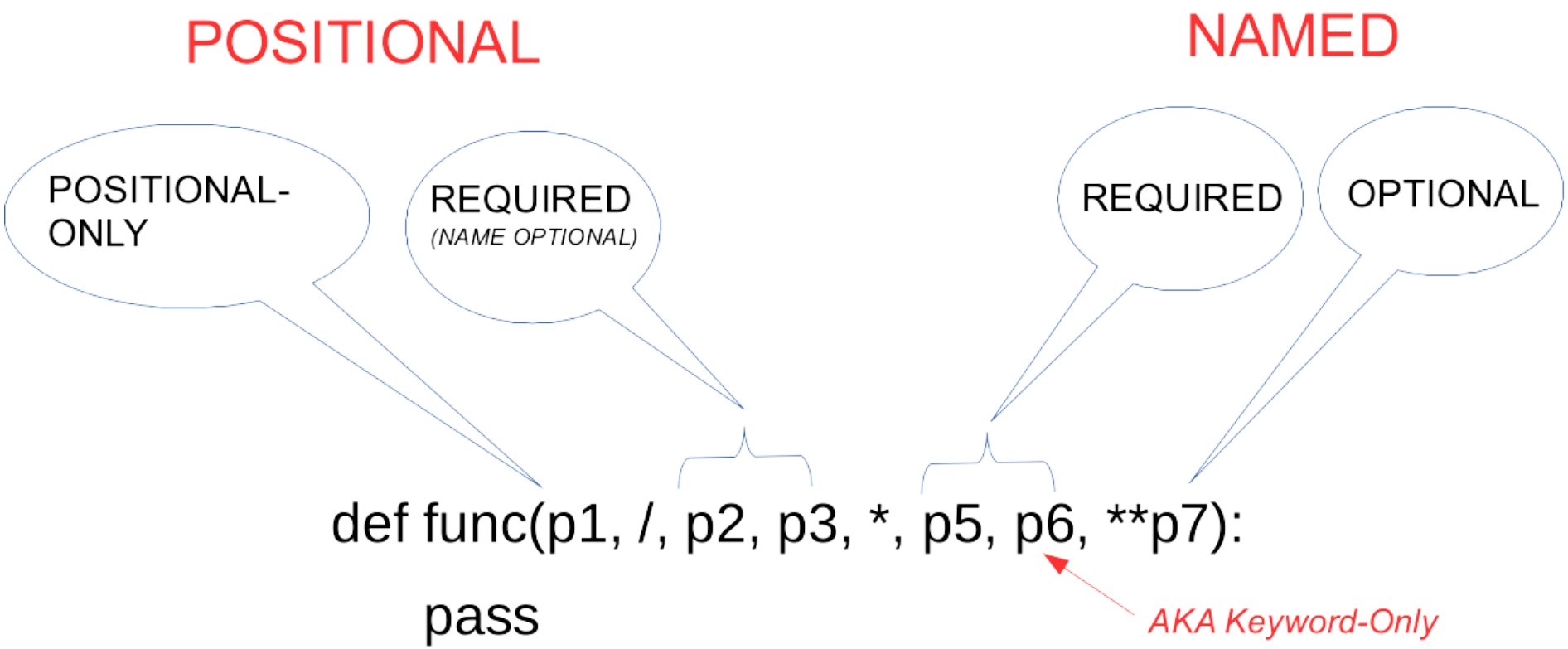
Function parameters



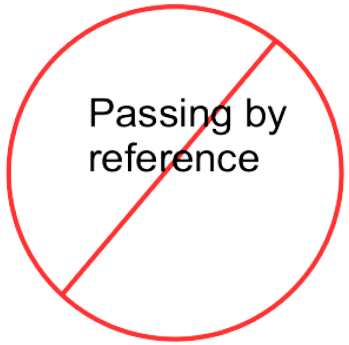
Function parameters



Function parameters



Argument passing

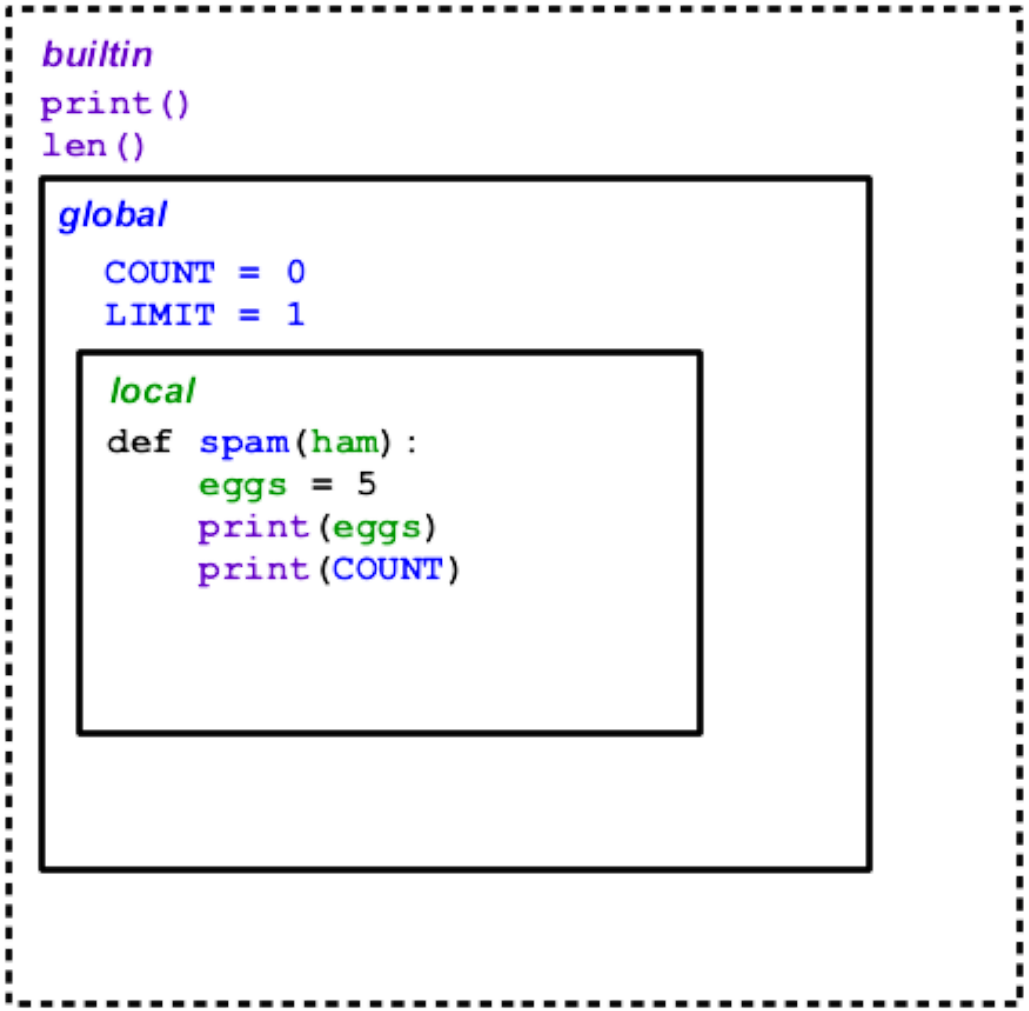


Passing by
sharing

- 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)
```

Variable Scope



Variable scope

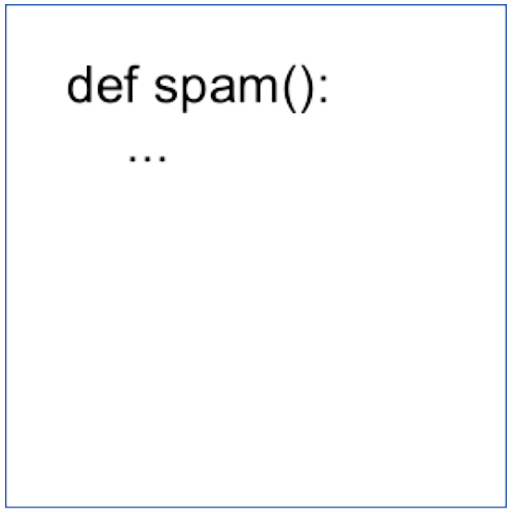
```
ALPHA = 10

def spam(beta):
    gamma = 20
    print(ALPHA)
    print(beta)
    print(gamma)

spam()
```

BUILTIN

GLOBAL

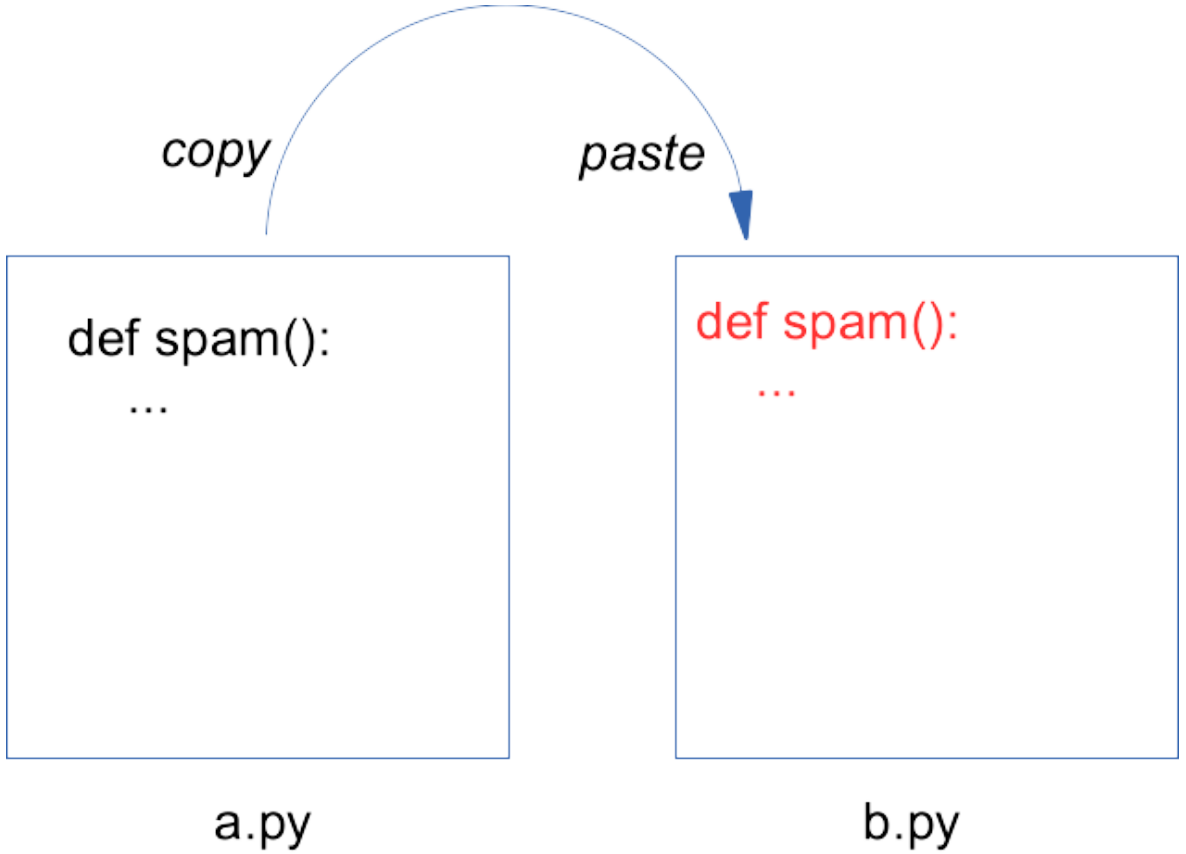
LOCAL

Copy/pasting functions

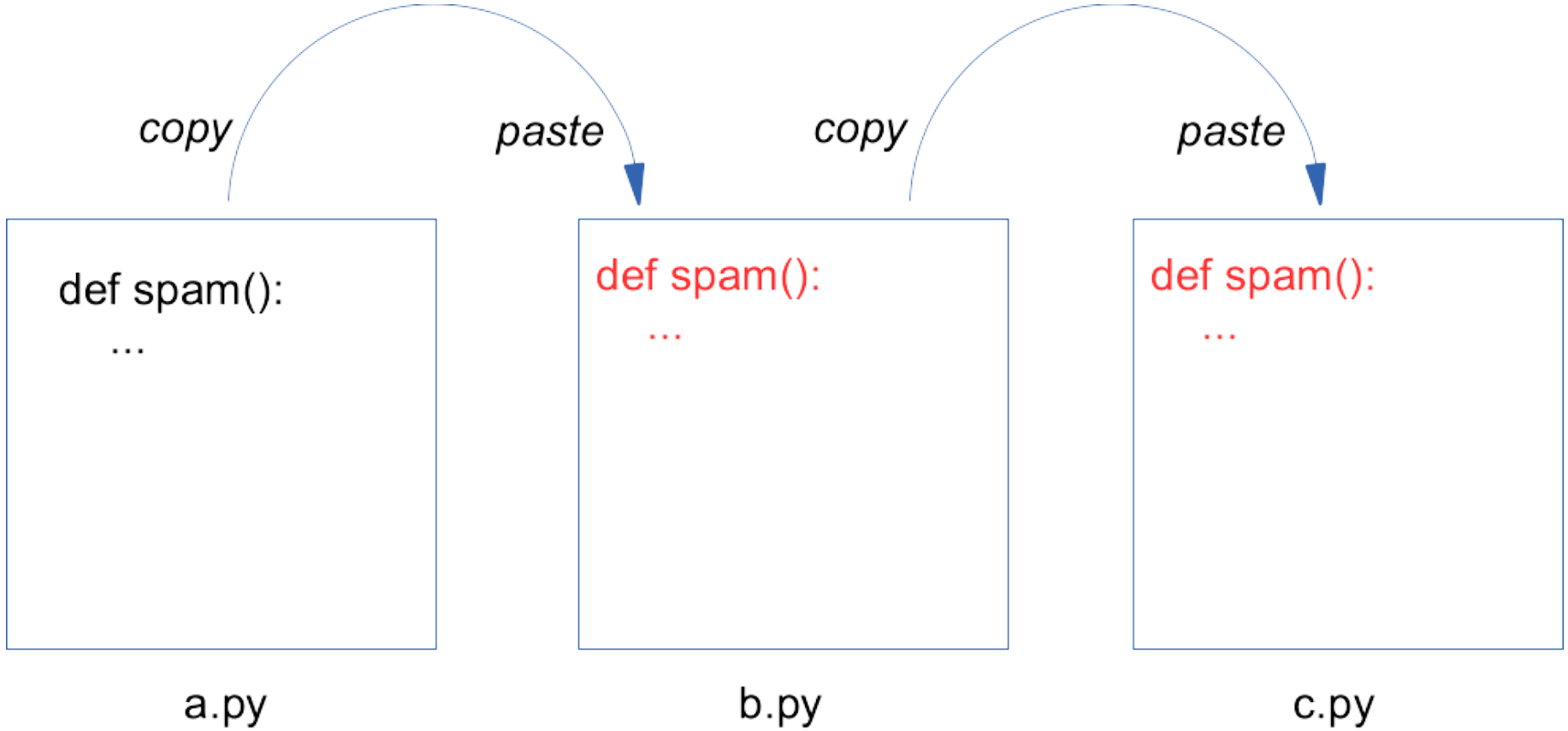


a.py

Copy/pasting functions

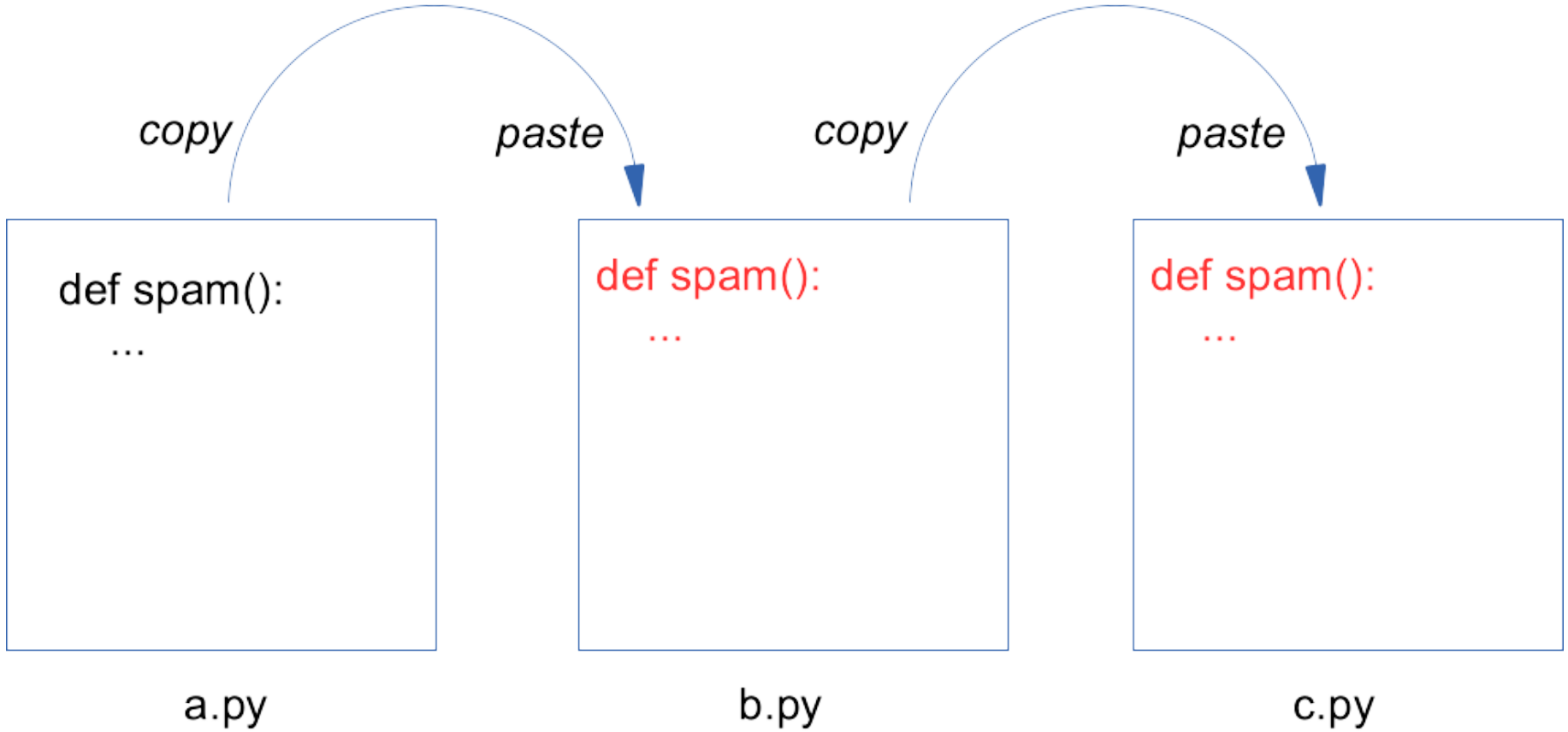


Copy/pasting functions

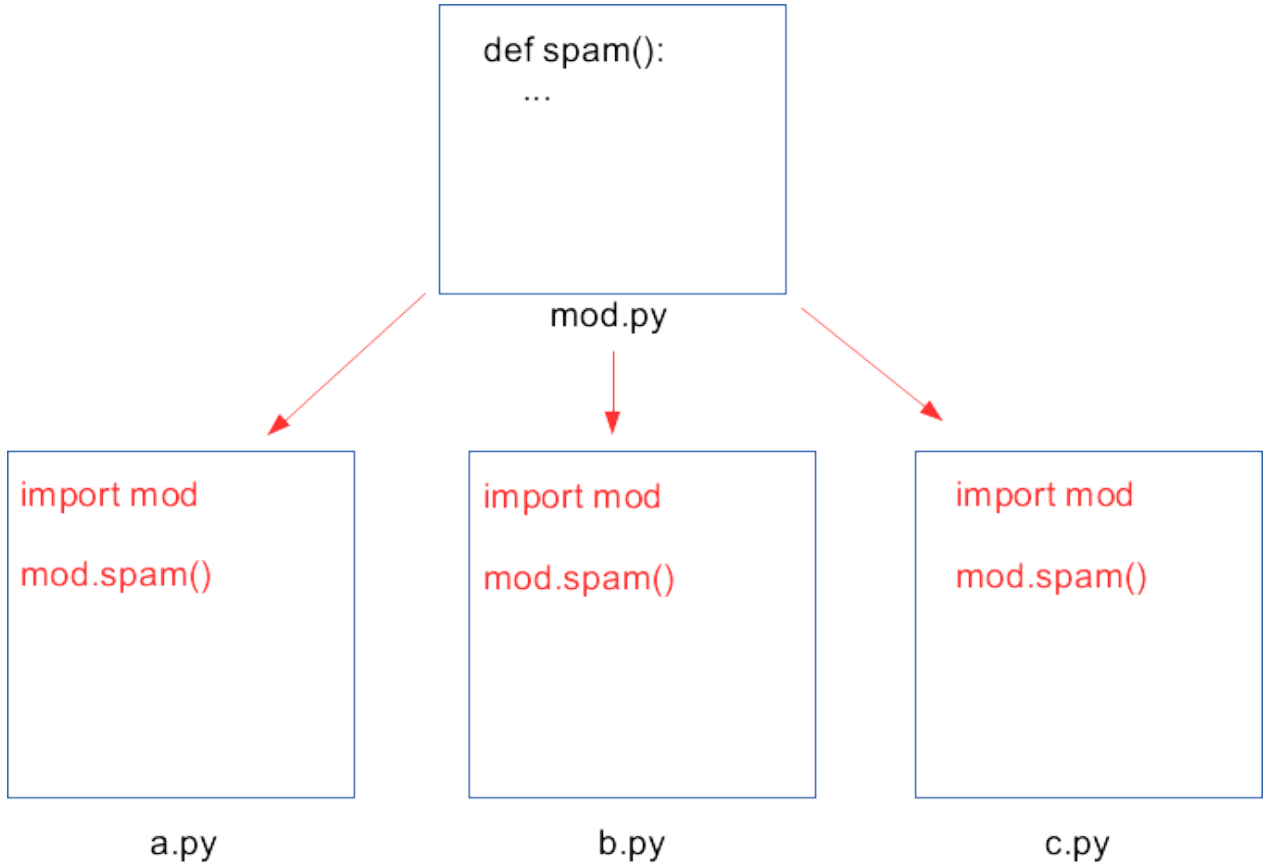


Copy/pasting functions

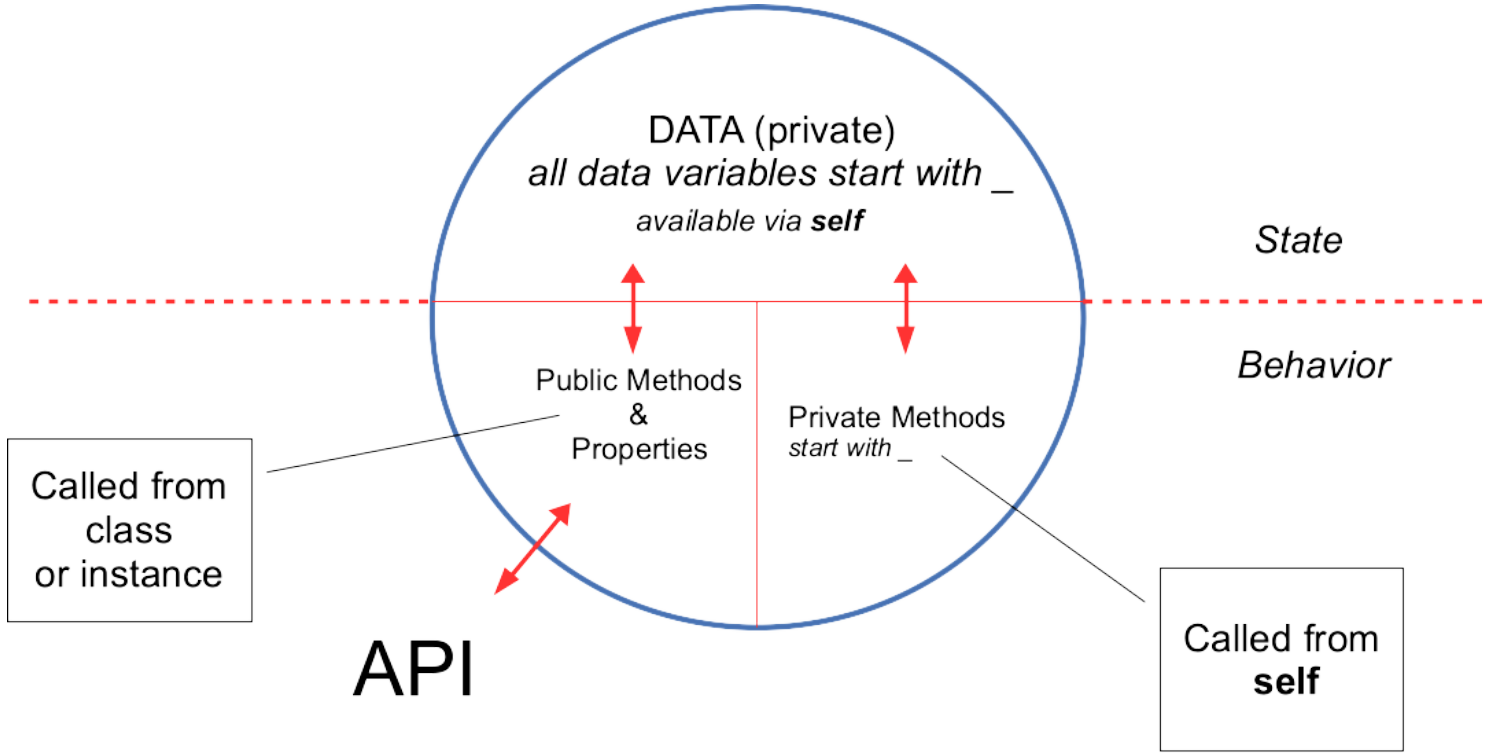
DON'T DO THIS!!



Using a module



A Python Class



Decorators Save Typing

Instead of

```
def spam():
    pass

spam = deco(spam)
```

use

```
@deco
def spam():
    pass
```

`spam` is only typed once, instead of 3 times

Decorator Syntax

```
@mydecorator
def some_function():
    pass
```

same as

```
some_function = mydecorator(some_function)
```

Implementation

```
def mydecorator(original_function):
    @wraps(original_function)
    def _(*args, **kwargs):
        # add code here
        result = original_function(*args, **kwargs)
        return result
    return _
```

Decorator with parameters

```
@mydecorator(param, ...)
def some_function():
    pass
```

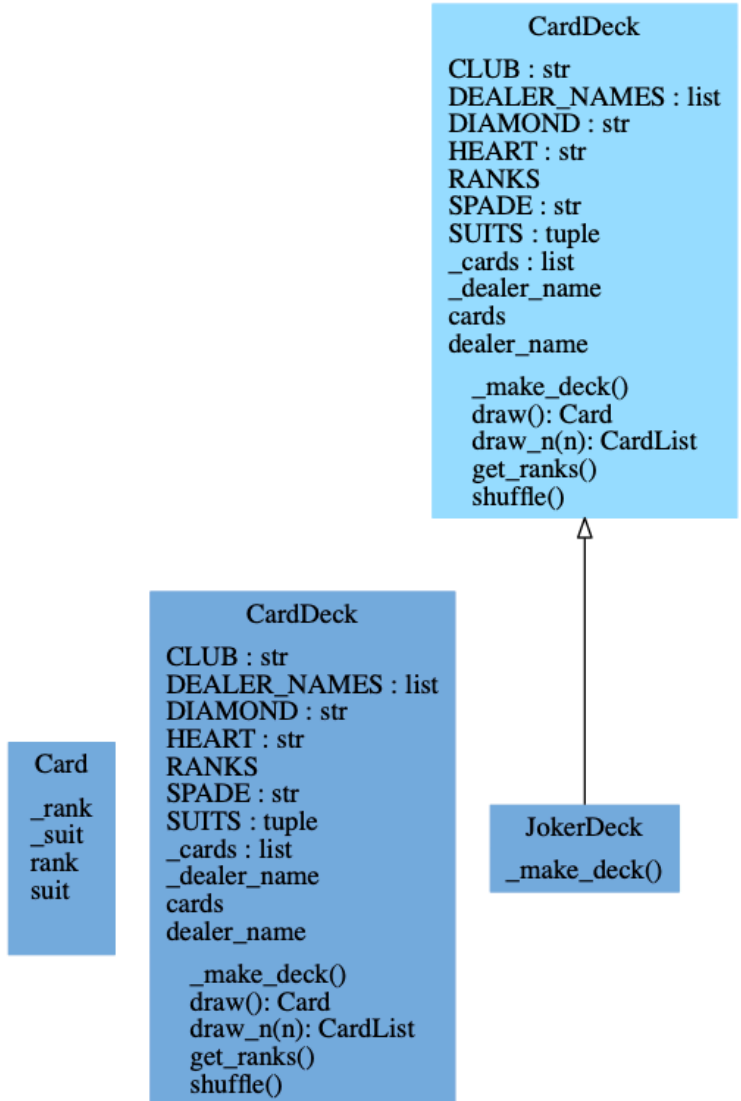
same as

```
some_function = mydecorator(param, ...)(some_function)
```

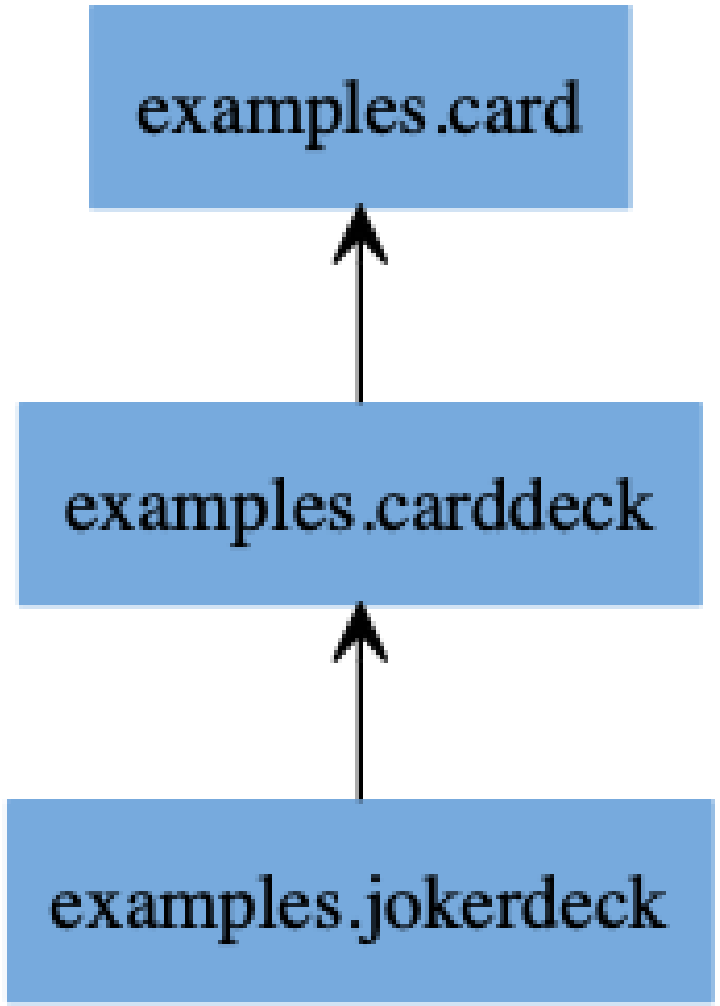
Implementation

```
def mydecorator(param, ...):
    def inner_decorator(original_function):
        @wraps(original_function)
        def _(*args, **kwargs):
            # add code here using decorator params
            result = original_function(*args, **kwargs)
            return result
        return _
    return inner_decorator
```

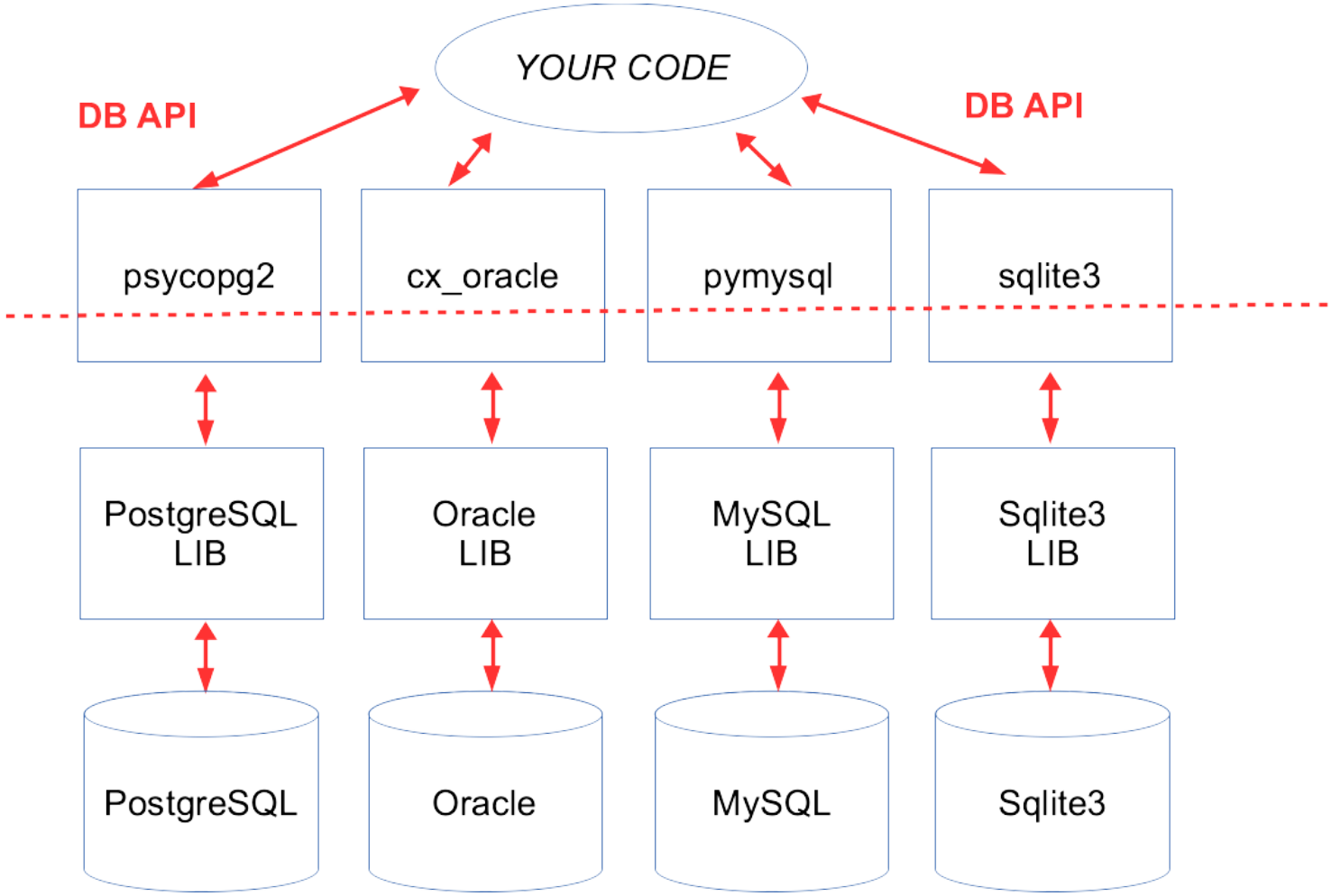
pyreverse (classes)



pyreverse (packages)



Python DB Interface



Python 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()

ElementTree

XML

```
<presidents>
    <president term="1">
        <first>George</first>
        <last>Washington</last>
    </president>
    <president term="2">
        <first>John</first>
        <last>Adams</last>
    </president>
</presidents>
```

ElementTree

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

Regular expression tasks

SEARCH

Is the match in the text?

RETRIEVE

Get the matching text

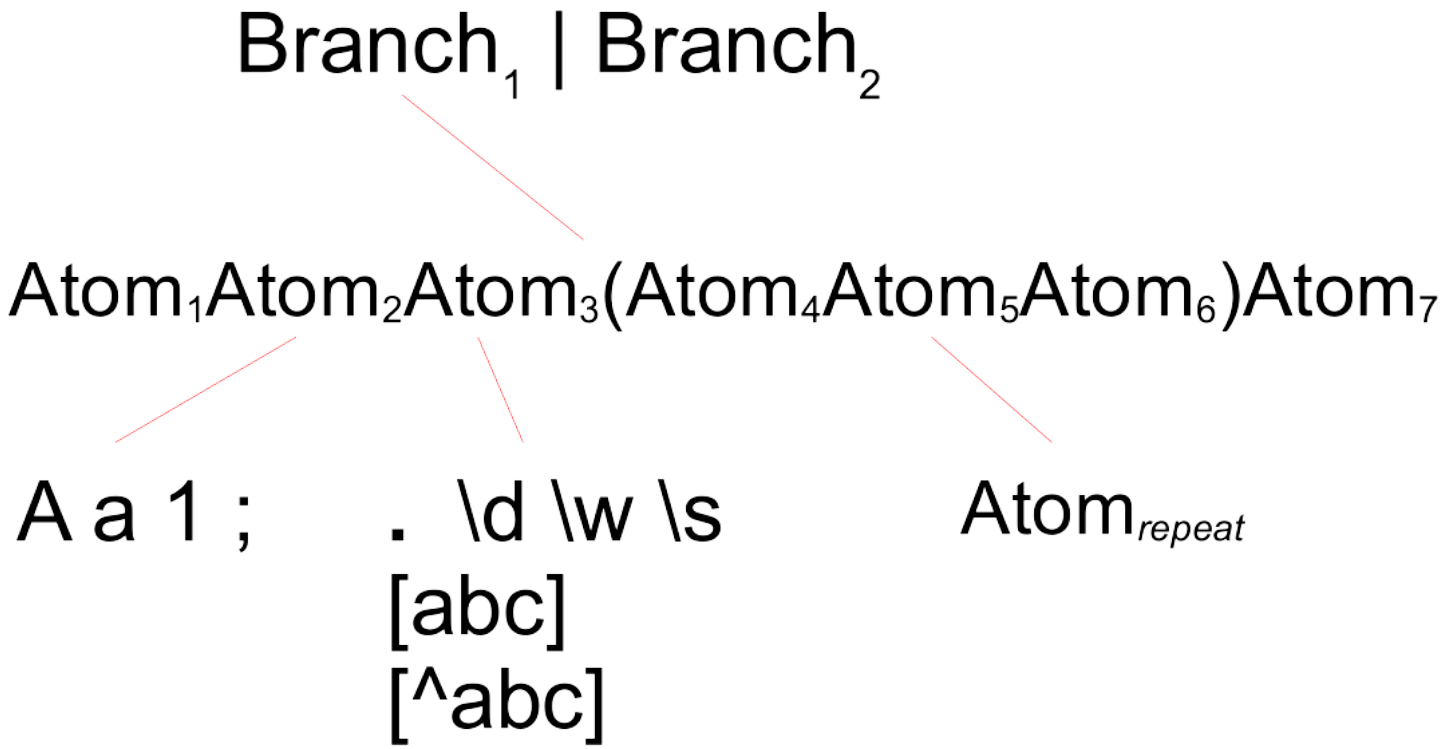
REPLACE

Substitute new text for match

SPLIT

Get what ***did not*** match

Regular Expression Components



Regular expression functions

- All functions take pattern and text
- Option flags can be added

Finding first match

`re.search(`*`pattern, text`*`)`

Find pattern and return **match** object

`re.match(`*`pattern, text`*`)`

Find pattern and return **match** object (implied ^*PATTERN*)

`re.fullmatch(`*`pattern, text`*`)`

Find pattern and return **match** object (implied ^*PATTERN*$)

Finding all matches

`re.finditer(`*`pattern`*`, `*`text`*`)`

Return iterable of **match** objects for all matches in text

`re.findall(`*`pattern`*`, `*`text`*`)`

Return list containing text of all matches

Replacing

`re.sub(`*`pattern, replacement, text`*`)`

Replace pattern with **replacement** and return new text

`re.subn(`*`pattern, replacement, text`*`)`

Replace pattern with **replacement** and return tuple with number of subs and new text

Splitting

`re.split(`*`pattern`*`, `*`text`*`)`

Split **text** using re as delimiter and return tokens as list.