Advanced Python Programming

TTPS4850

Creating Variables

```
x = 5
```

Creating Variables

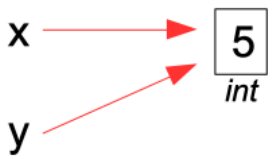
```
x = 5
```

Creating Variables

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

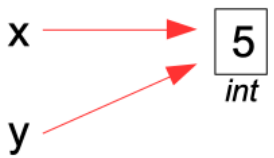
Creating Variables

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



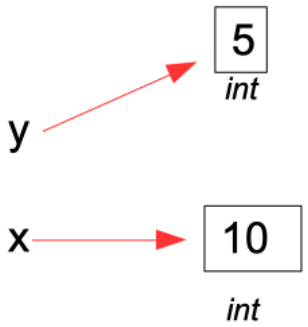
Creating Variables

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



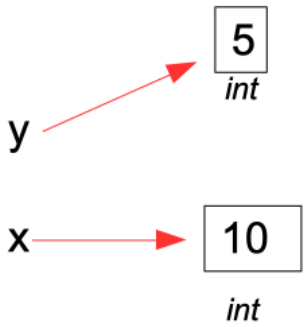
Creating Variables

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



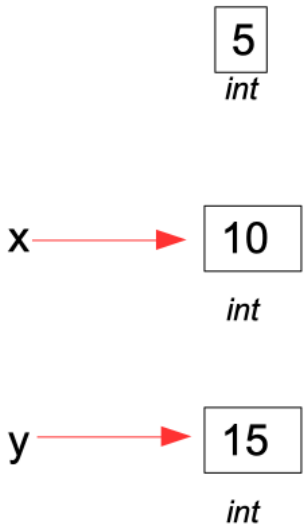
Creating Variables

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



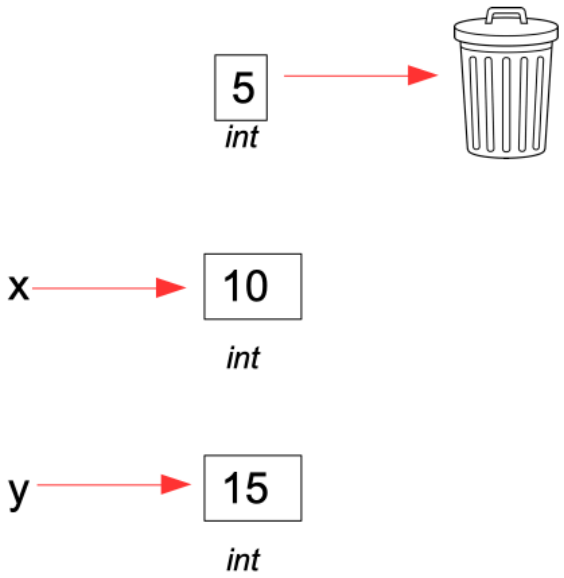
Creating Variables

```
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

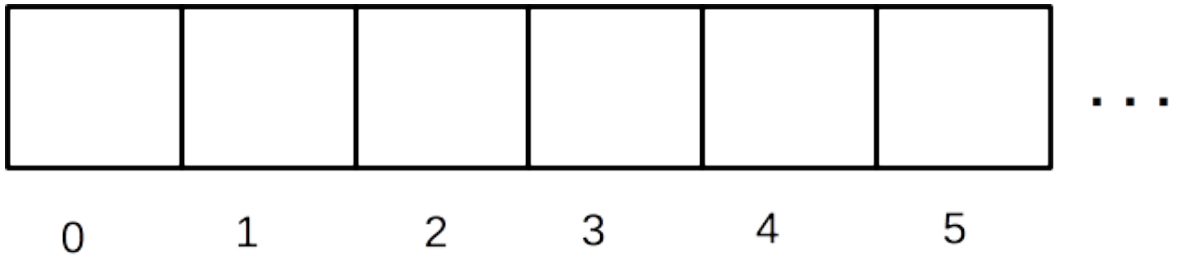
```
"""spam\n"""
'''spam\n'''

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'
```

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"
```

Dictionary

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

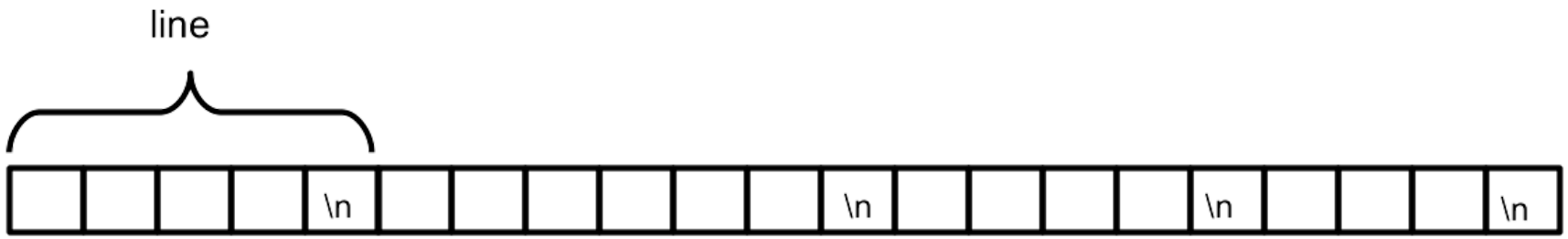
Dictionary items

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

Reading Text Files

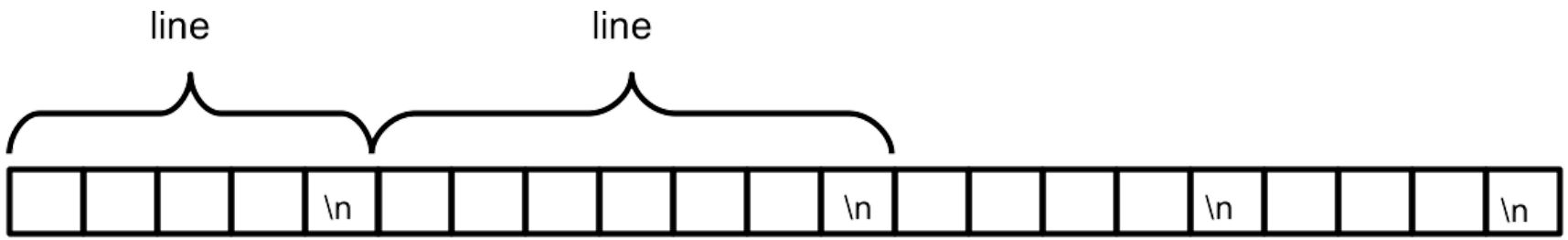

```
with open("somefile") as file_in:
```

Reading one line at a time



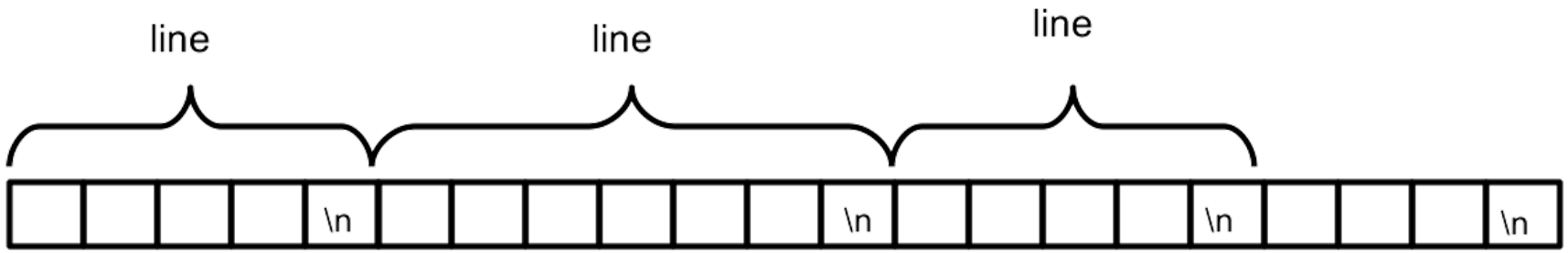

```
with open("somefile") as file_in:
    for raw_line in file_in:
        ...
```

Reading one line at a time



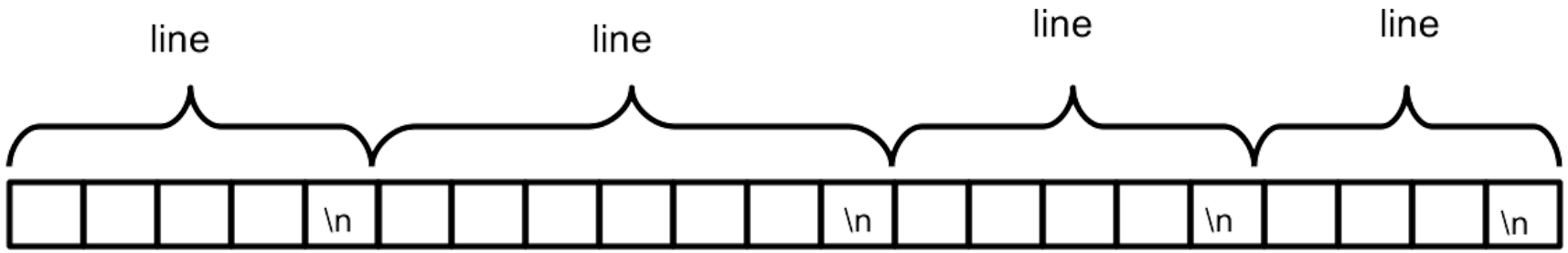

```
with open("somefile") as file_in:
    for raw_line in file_in:
        ...
```

Reading one line at a time



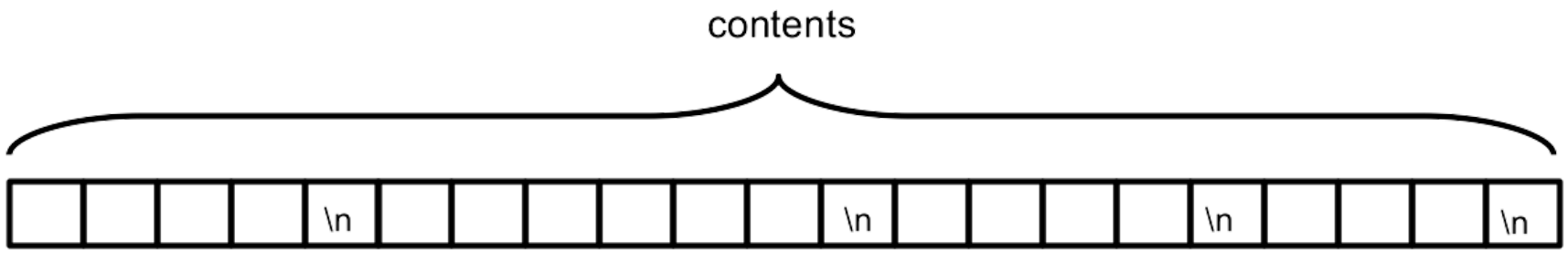

```
with open("somefile") as file_in:
    for raw_line in file_in:
        ...
```

Reading one line at a time



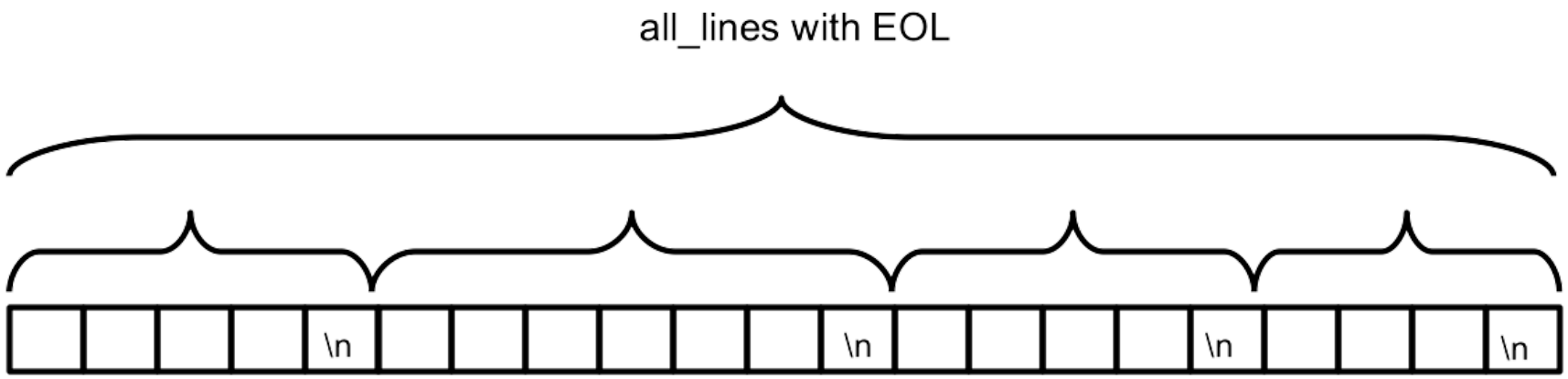

```
with open("somefile") as file_in:
    for raw_line in file_in:
        ...
```

Reading entire file into string



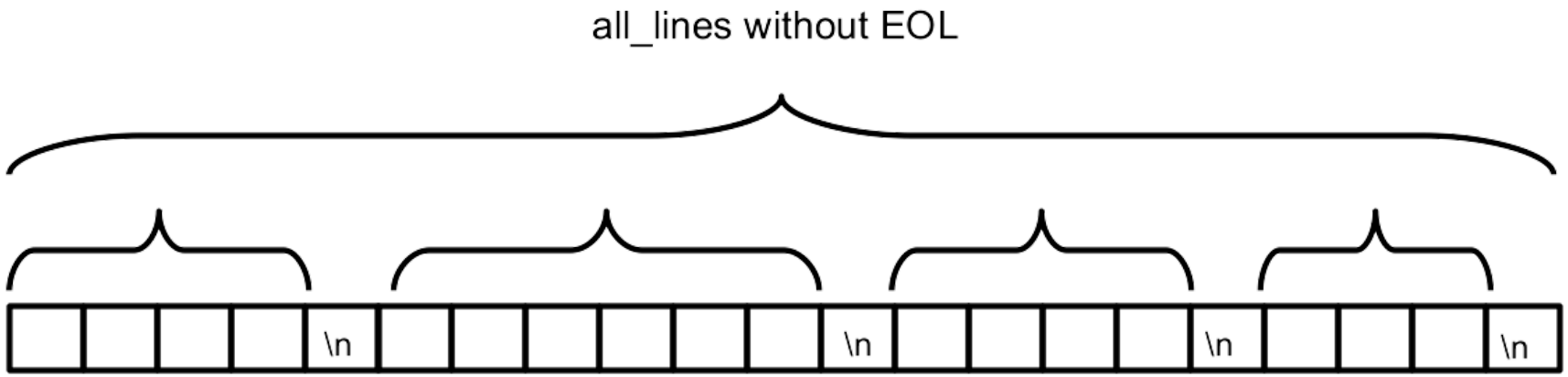

```
with open("somefile") as file_in:
    contents = file_in.read()
```

Reading file into list of strings (with EOL)



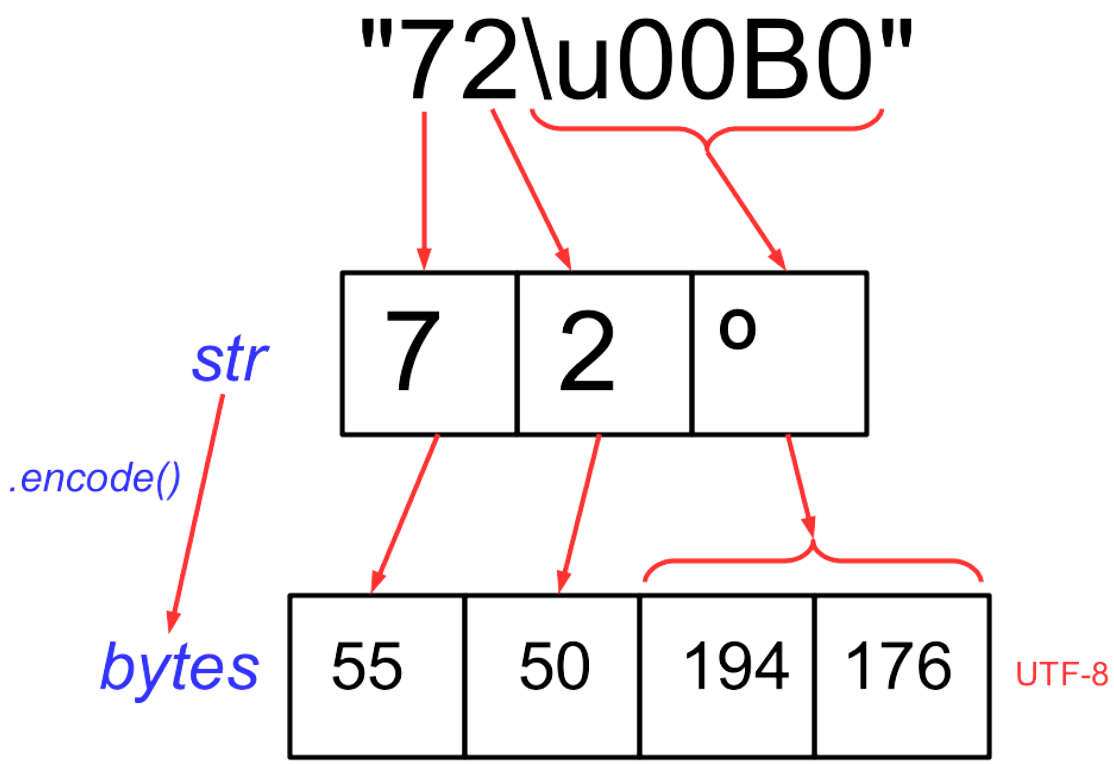

```
with open("somefile") as file_in:
    all_lines = file_in.readlines()
```

Reading file into list of strings (without EOL)

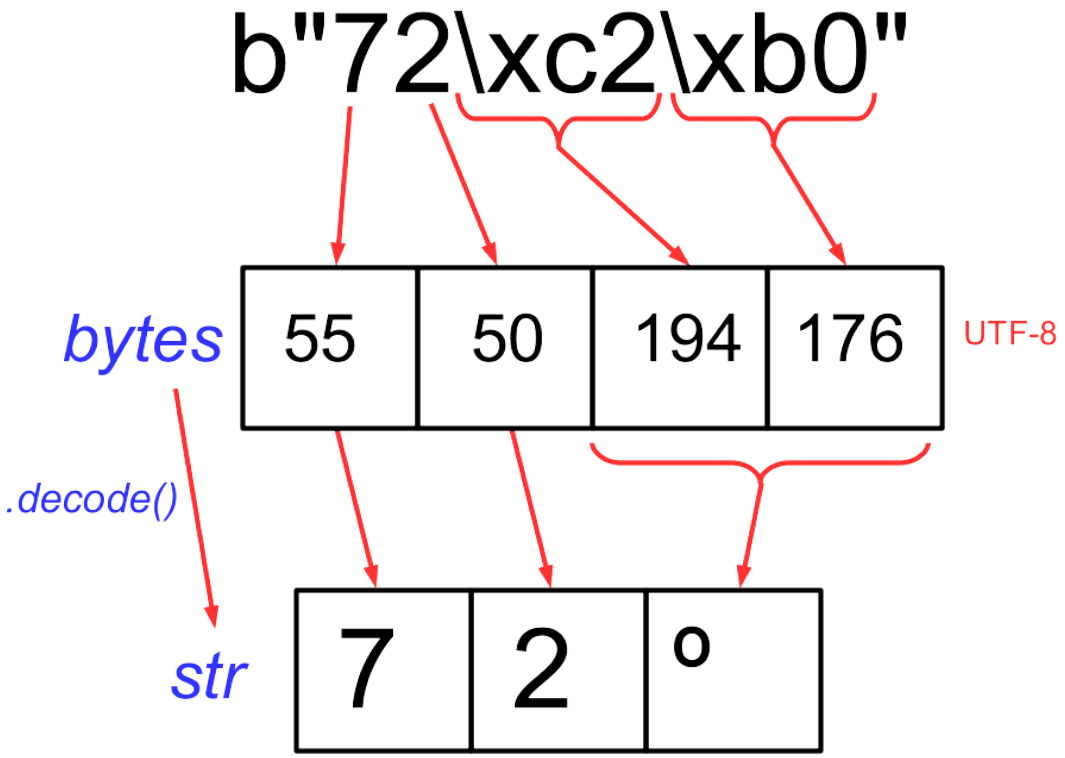



```
with open("somefile") as file_in:
    all_lines = file_in.read().splitlines()
```

str to bytes



bytes to str



Lists vs Tuples

Lists	Tuples
Dynamic array	Collection of related fields
Mutable/unhashable	Immutable/hashable
Position doesn’t matter	Position matters
Use case: iterating	Use case: indexing or unpacking
"ARRAY"	"STRUCT" or "RECORD"

A Myth

Tuples are just read-only lists

Tuple alternatives

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

Sorting

Numbers

`n, n, n, …`

Strings

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

Nested iterables

[Obj_1, Obj_2, Obj_3], [Obj_1, Obj_2, Obj_3],

Dictionary elements

(*key*, *value*), (*key*, *value*), (*key*, *value*),

Sequence Comprehensions

- list comprehension

```
[EXPR for VAR ... in ITERABLE if CONDITION]
```

- generator expression

```
(EXPR for VAR ... in ITERABLE if CONDITION)
```

Mapping Comprehensions

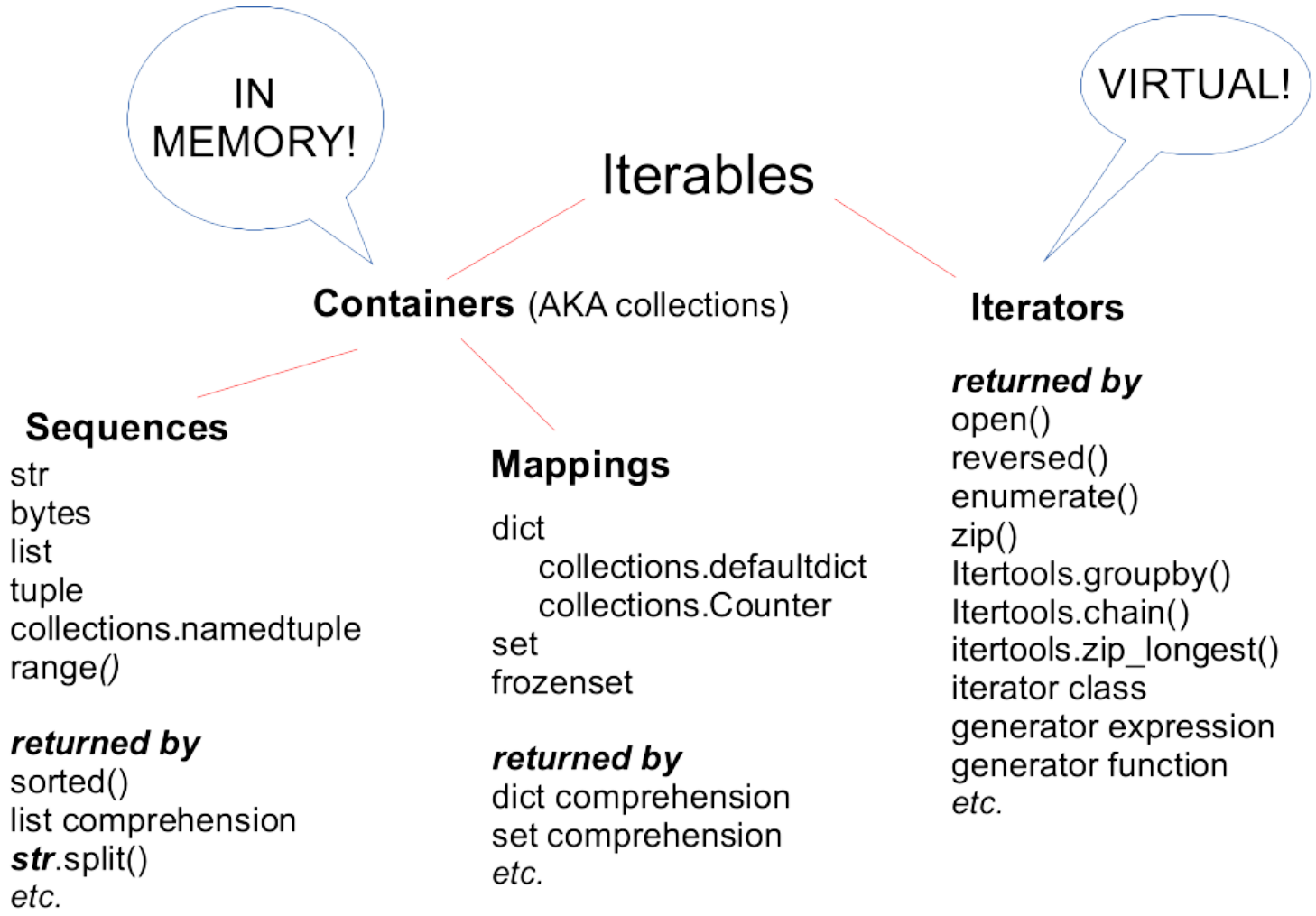
- dict comprehension

```
{KEY-EXPR: VALUE-EXPR for VAR ... in ITERABLE if CONDITION}
```

- set comprehension

```
{EXPR for VAR ... in ITERABLE if CONDITION}
```

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`

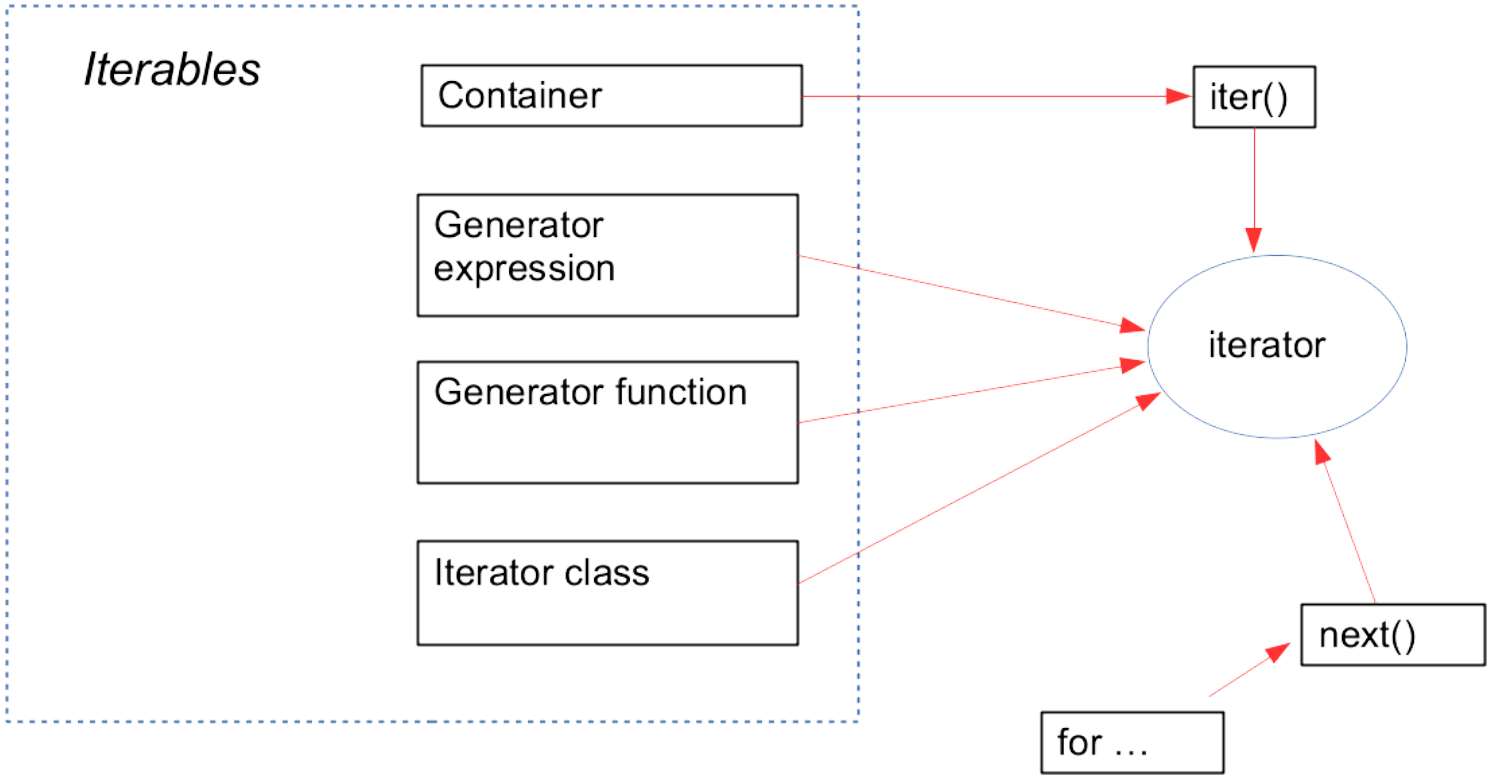
Iterators

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

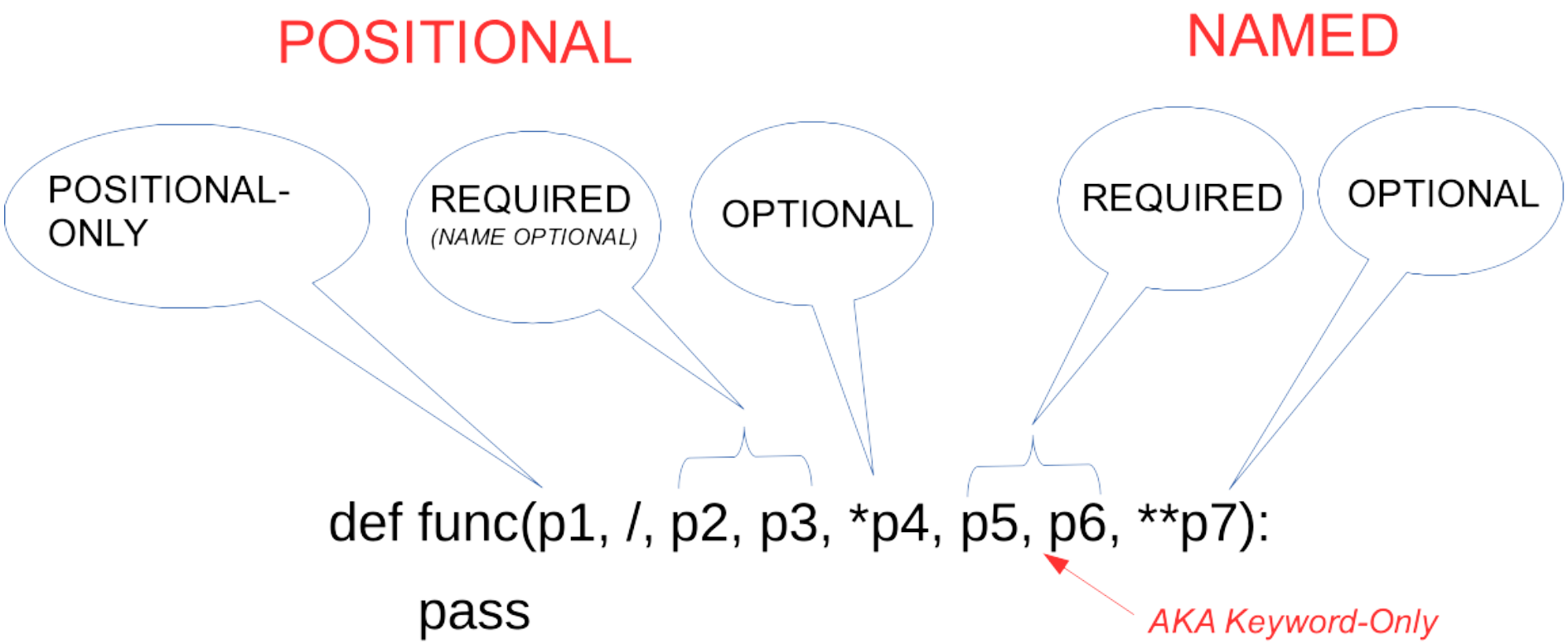
Iterators returned by

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

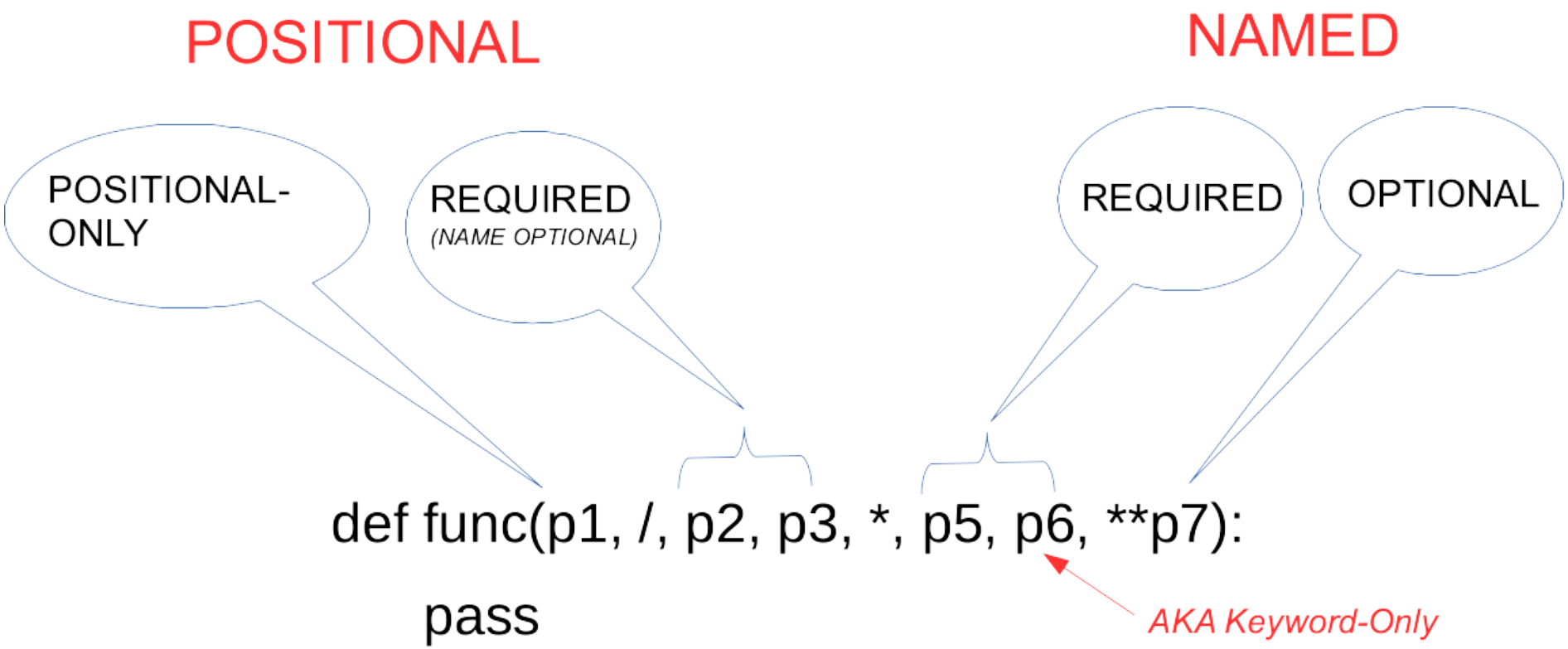
Iterators


Iterables
Container
Generator expression
Generator function
Iterator class
iter()
iterator
next()
for …

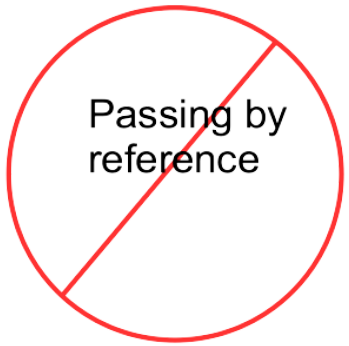
Function parameters



Function parameters



Argument passing

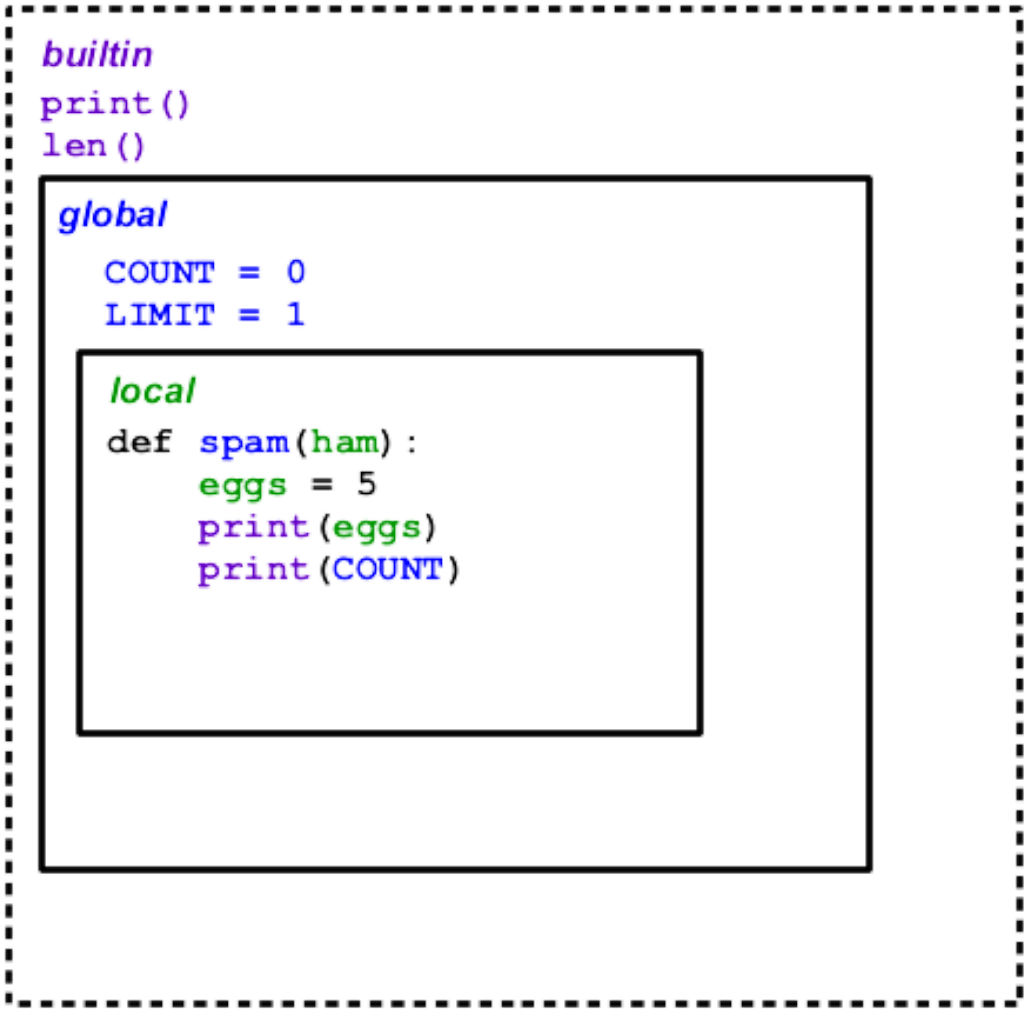


Passing by
sharing

- 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(1234)
```

BUILTIN

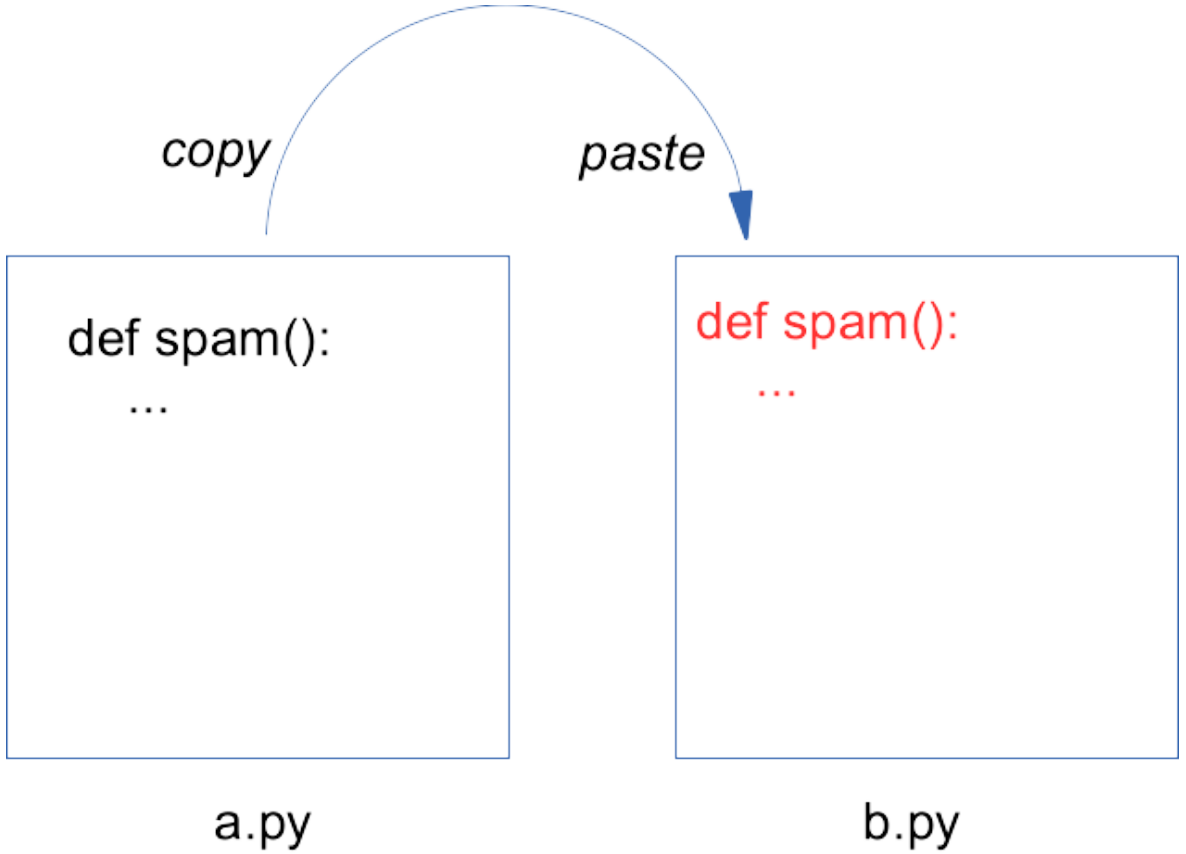
GLOBAL

LOCAL

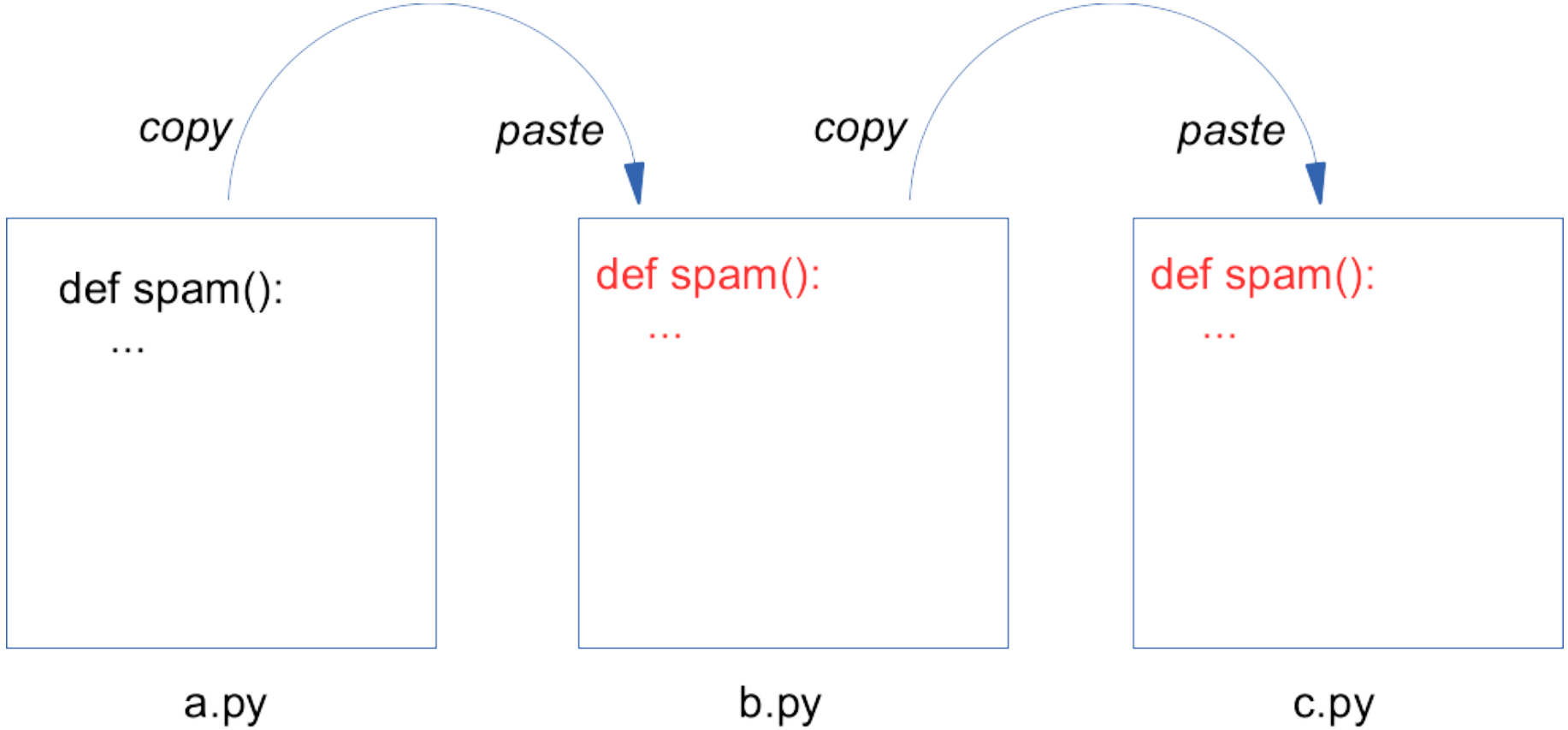
Copy/pasting functions

a.py

Copy/pasting functions

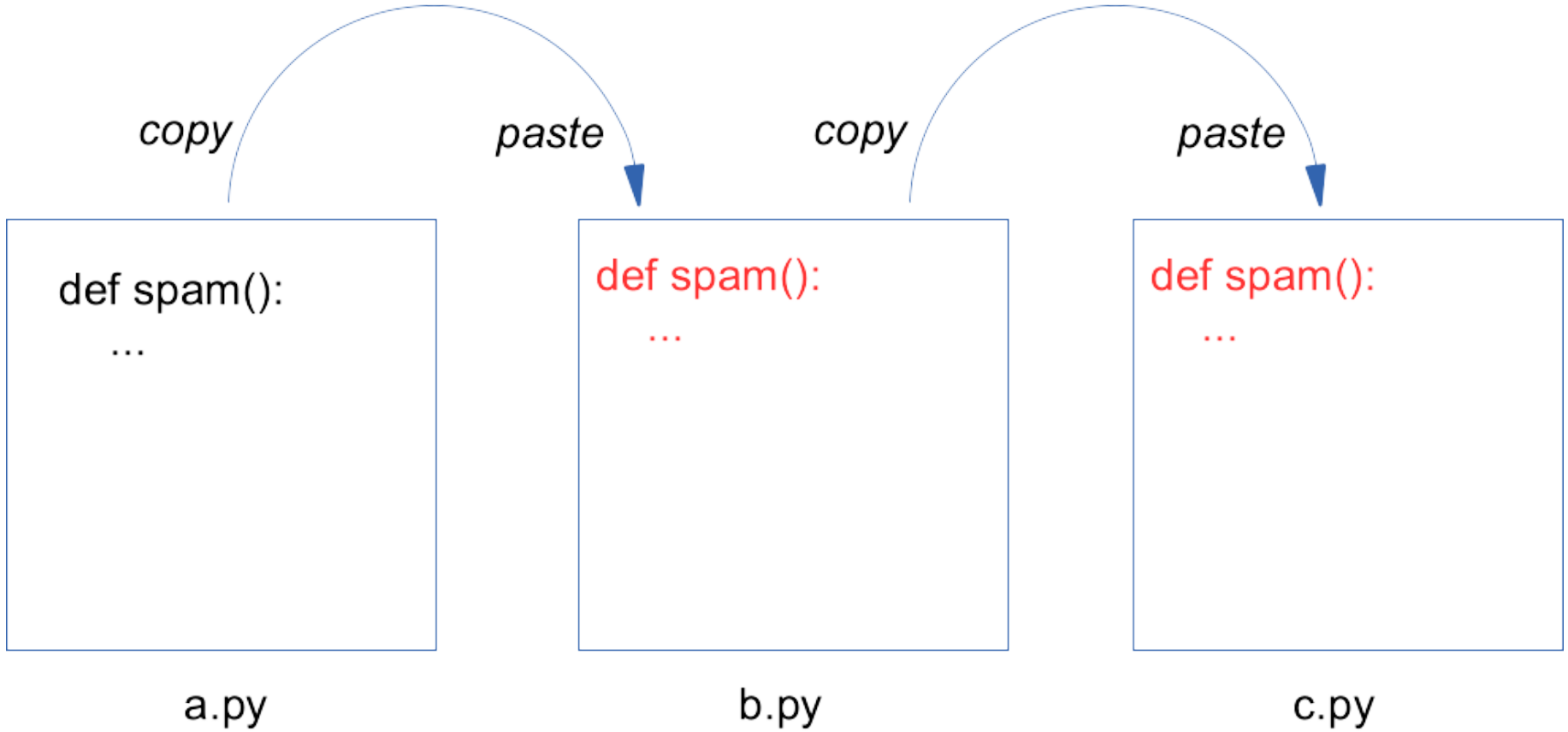


Copy/pasting functions

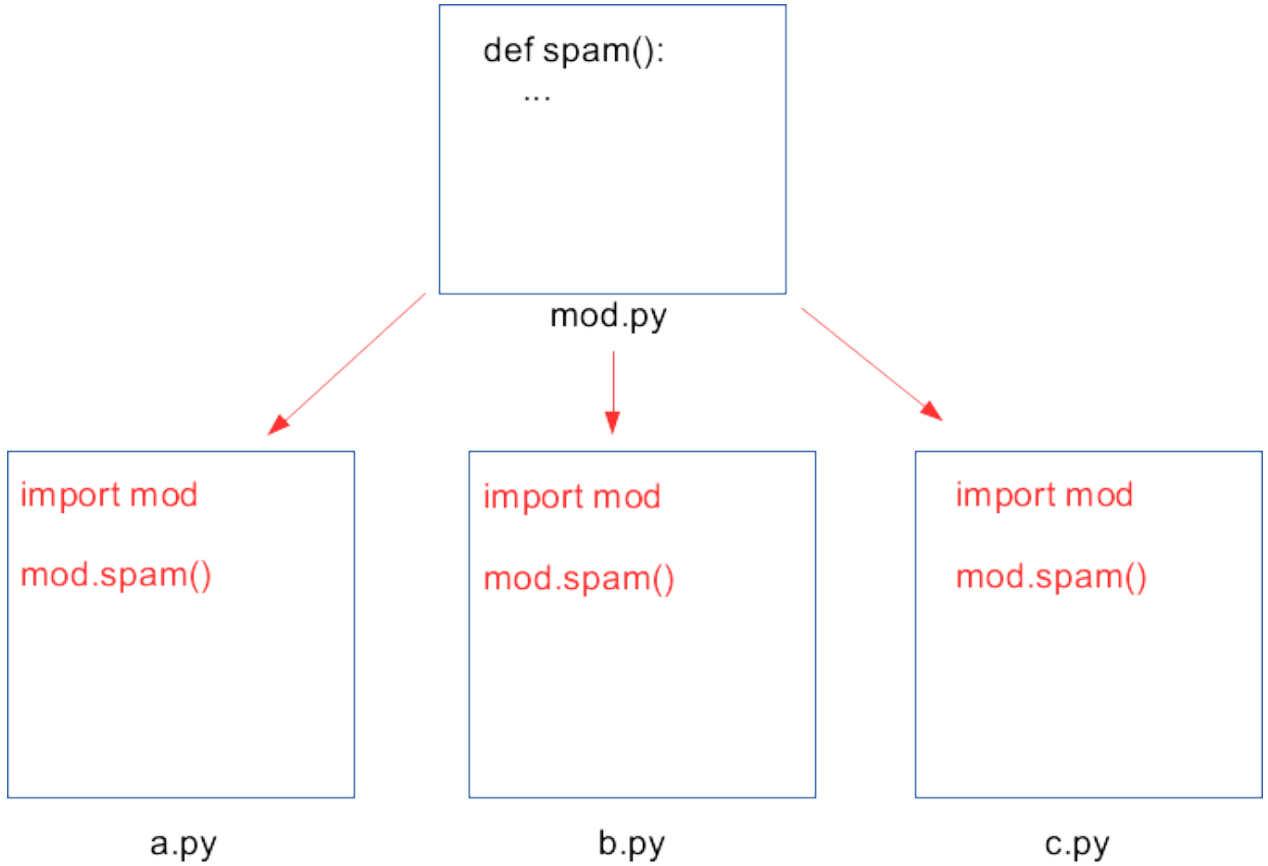


Copy/pasting functions

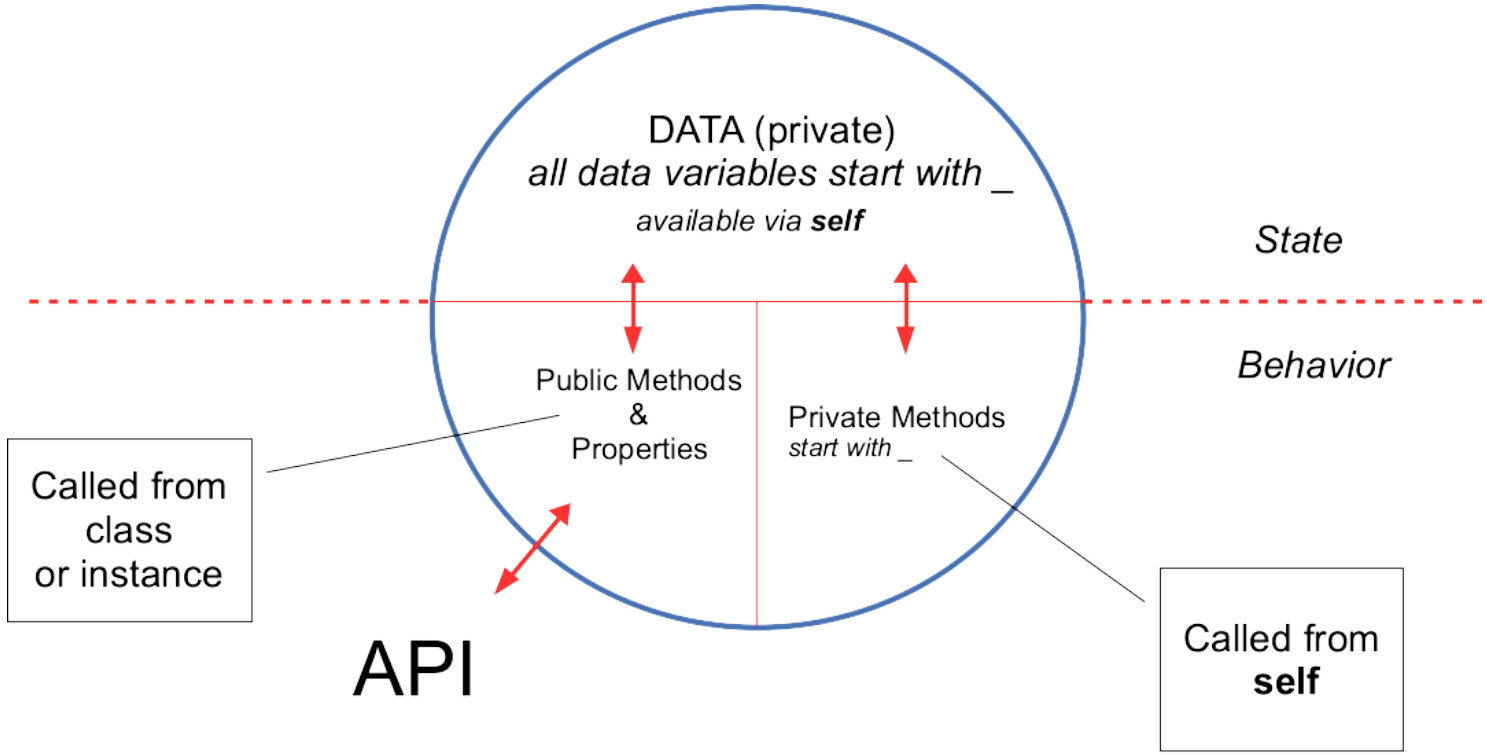
DON'T DO THIS!!



Using a module



A Python Class



`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()`	If undefined, uses `object.__repr__()`

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

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

same as

```
some_function = DECORATOR(some_function)
```

Implementation

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

Decorator with parameters

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

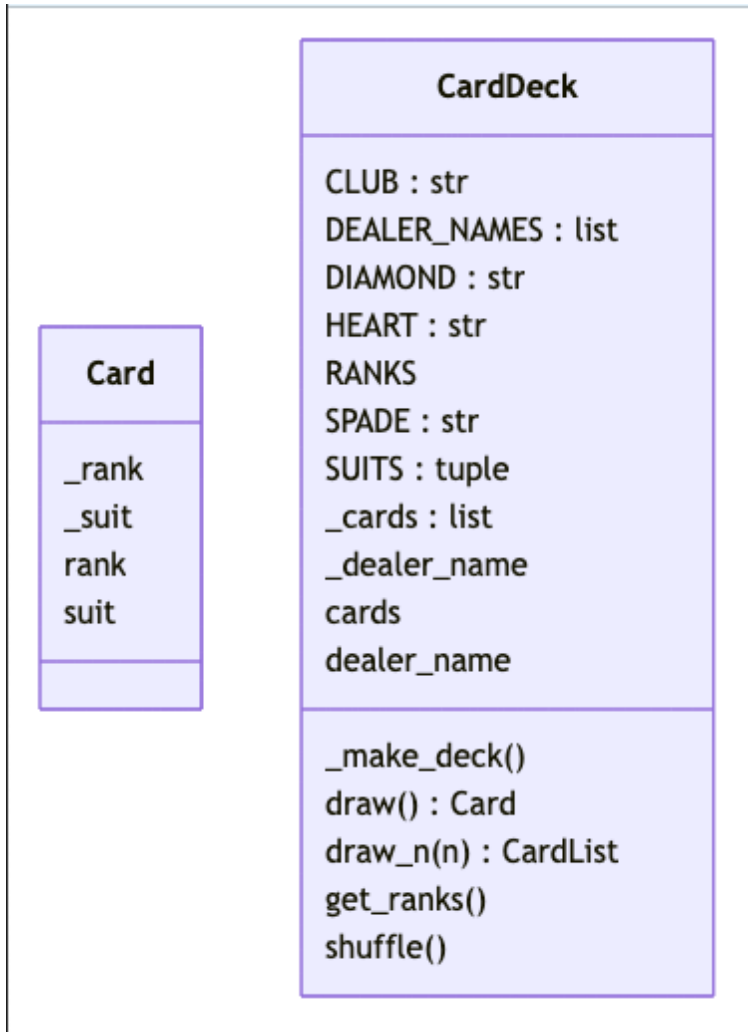
same as

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

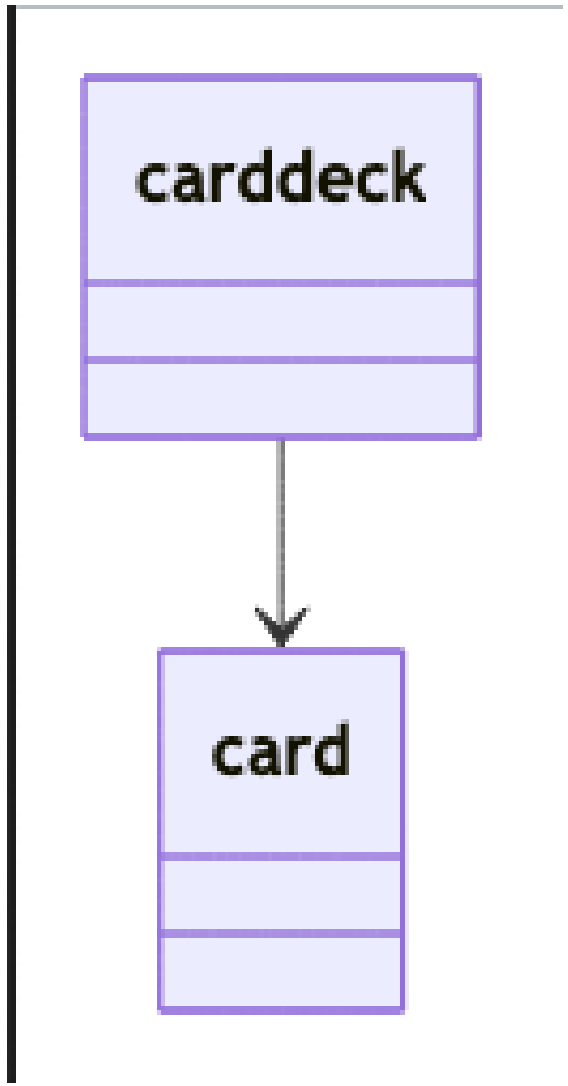
Implementation

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

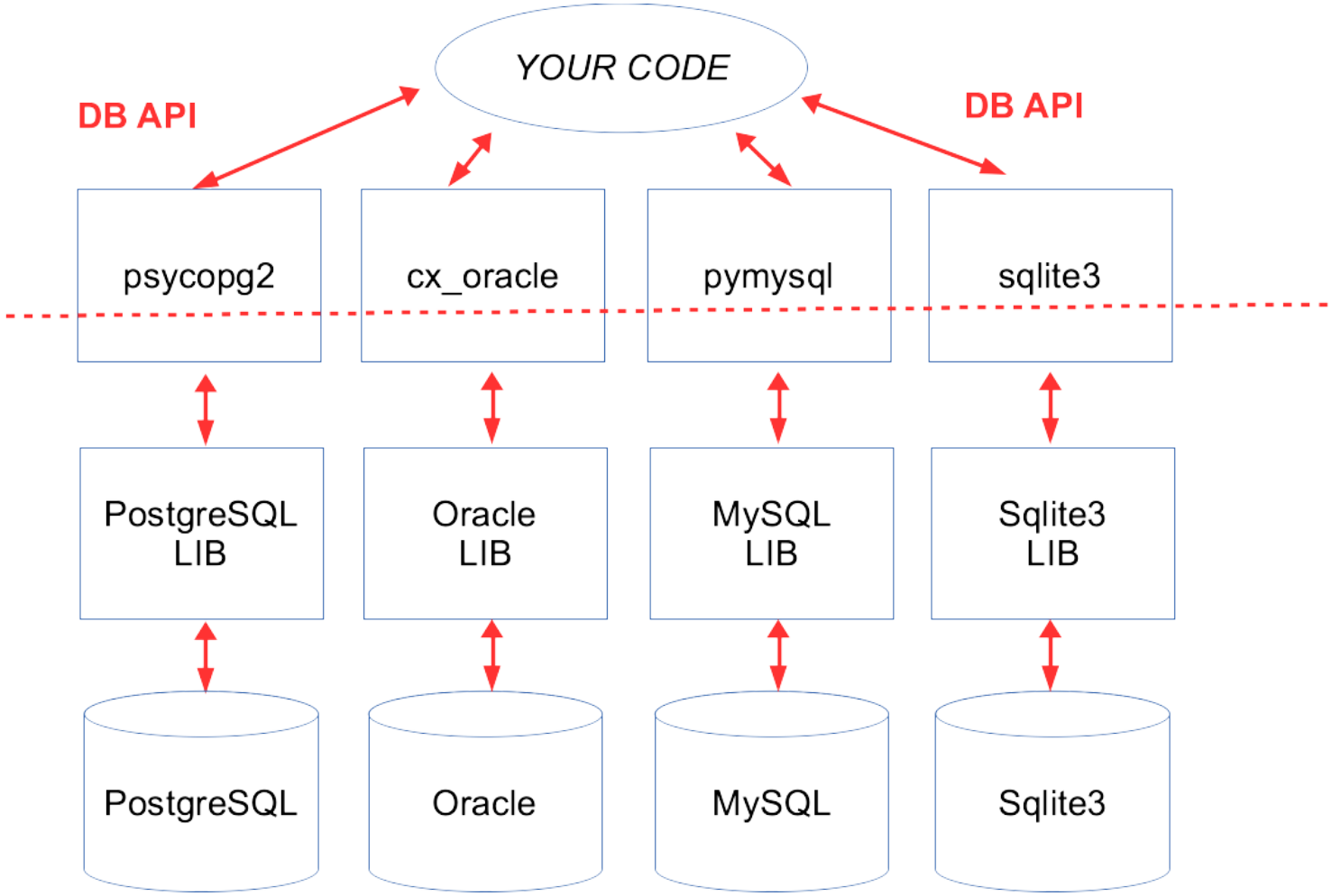
pyreverse (classes)



pyreverse (packages)



Python DB Interface



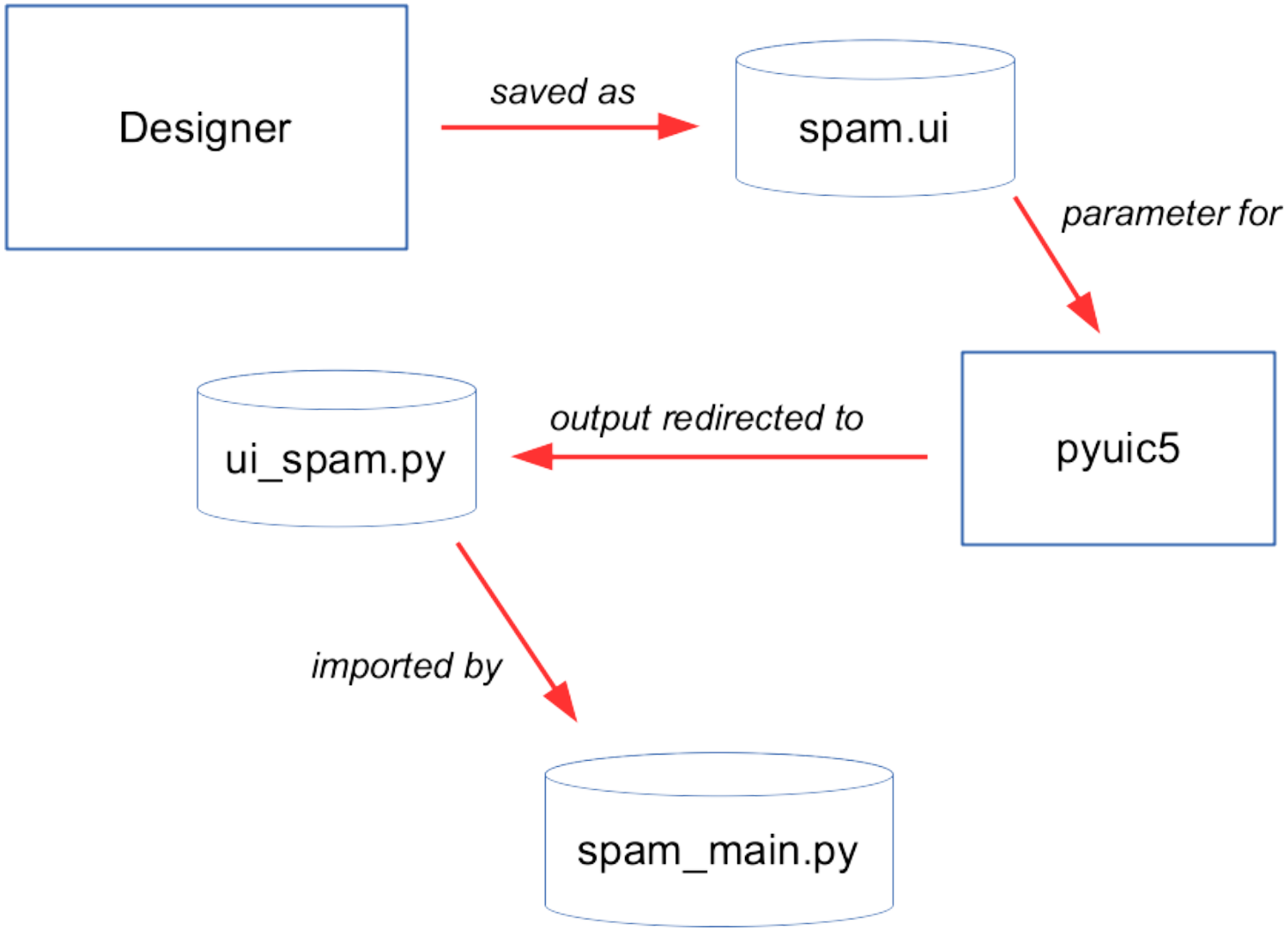
Python DB API

- 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()

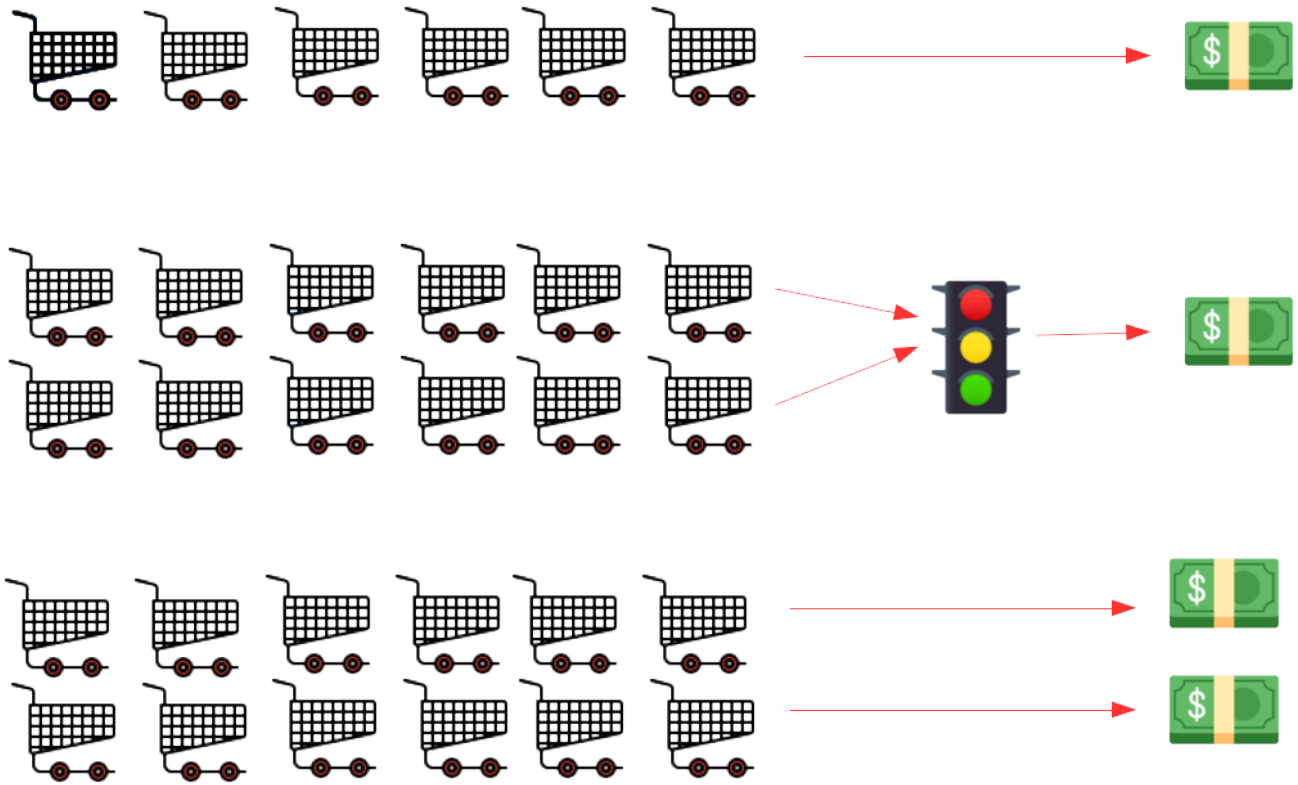
testing

```
conn* = package.connect(server, db, user, password, etc.)`
    conn.commit()
    conn.rollback()
cursor = conn.cursor()
    cursor.execute()
    cursor.executemany()
    cursor.executescript()
    cursor.fetchall()
    cursor.fetchone()
    cursor.fetchmany()
```

PyQt Designer Workflow



Concurrency



ElementTree

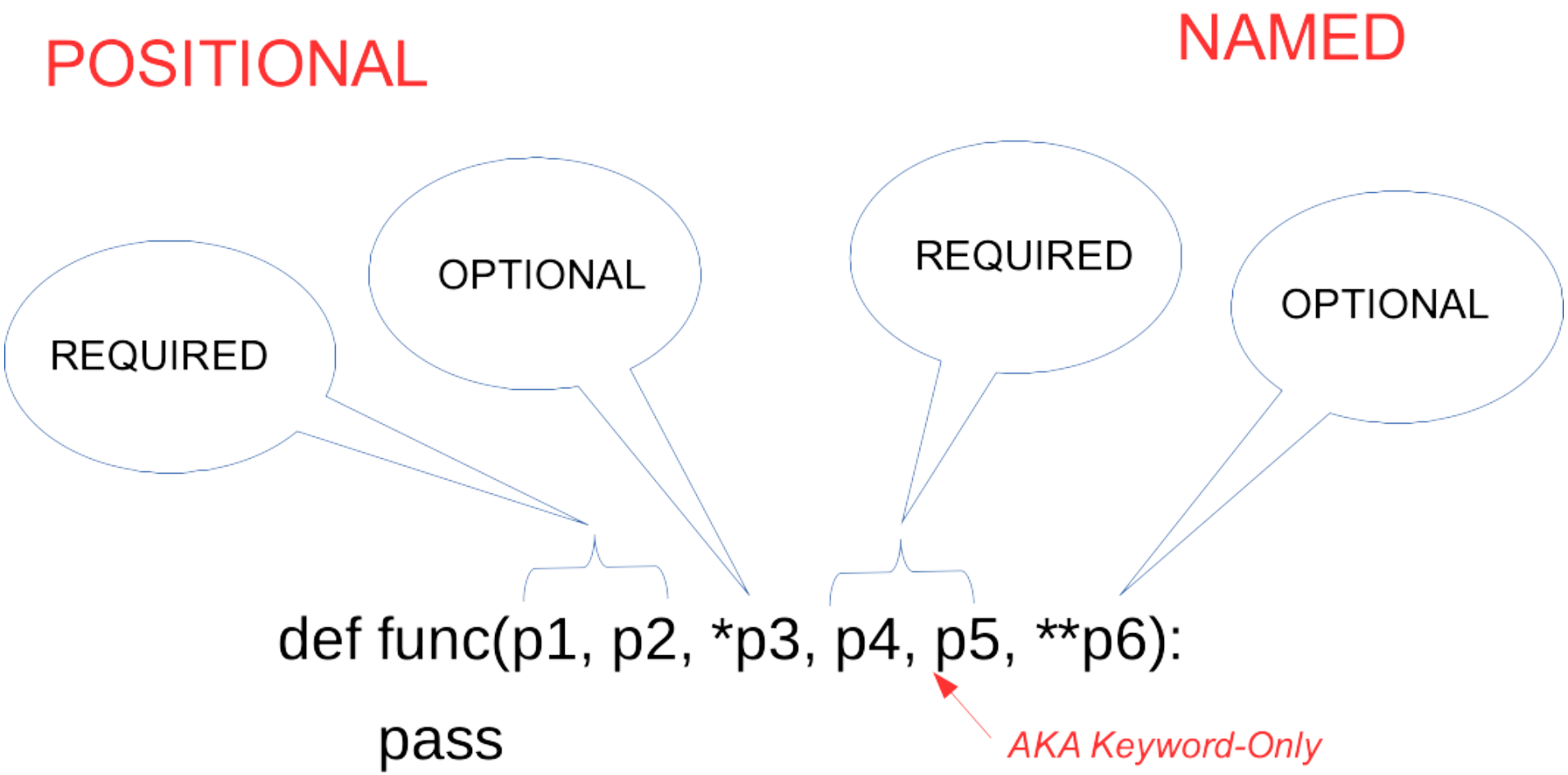
XML

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

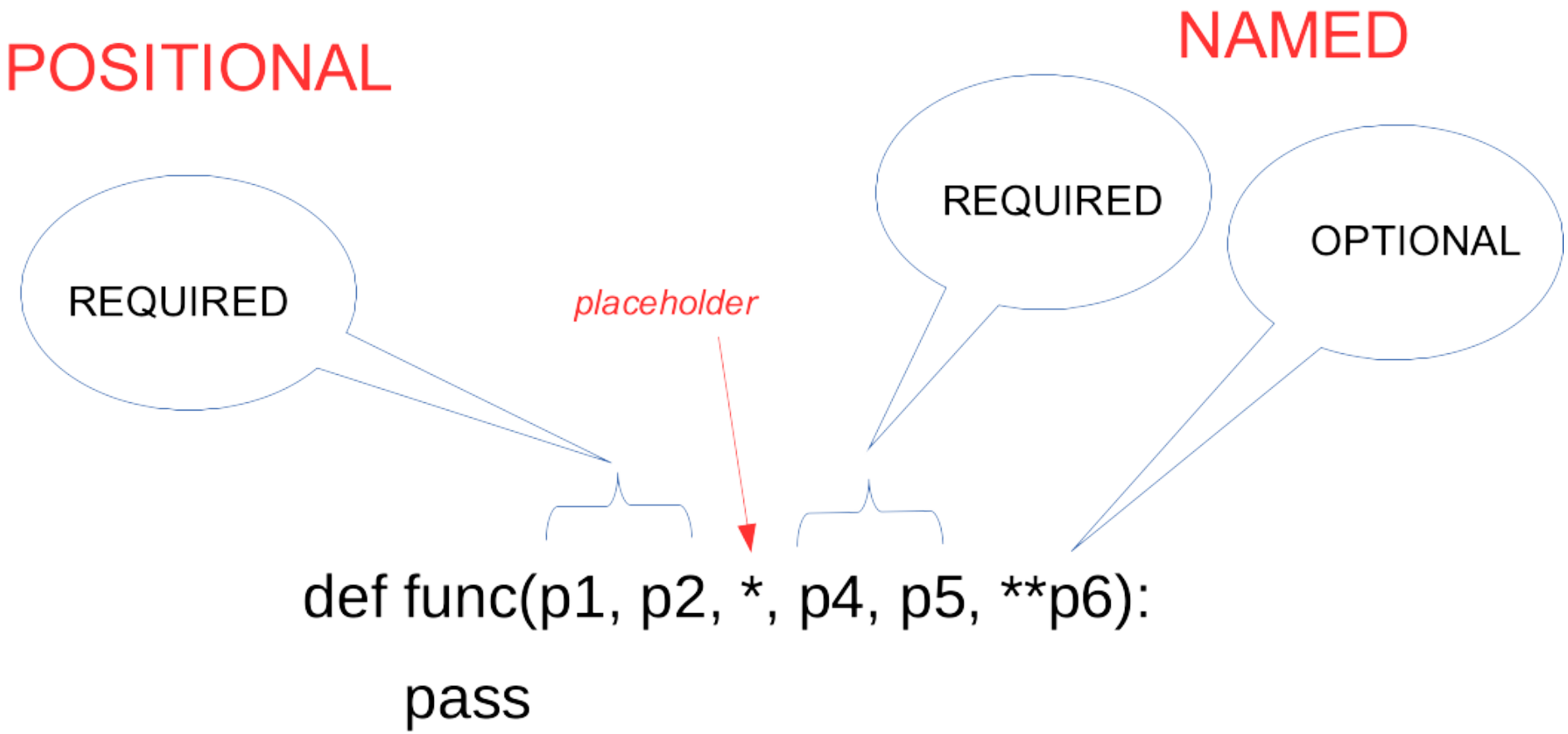
ElementTree

```
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"
```

Function parameters



Function parameters



Configuring Visual Studio code

Some settings to make programming with Python easier

Auto-save

- Search for "auto save"
- Set to *after delay*

Launch folder

- Search for "execute in"
- Check box for **Python > Terminal: Execute in File Dir**

Minimap

- Search for "minimap enabled"
- Uncheck **Editor > Minimap: Enabled**

Editor font size

- Search for "editor font size"
- Set **Editor: Font Size** to desired size

Terminal font size

- Search for "terminal font size"
- Set **Terminal: Font Size** to desired size

Themes

- Got to **File > Preferences > Theme > Color Theme**
- Select new theme as desired