Python Programming

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

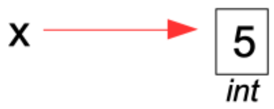
- 300+ modules
- Always available

Creating Variables

```
x = 5
```

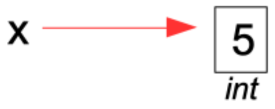
Creating Variables

```
x = 5
```



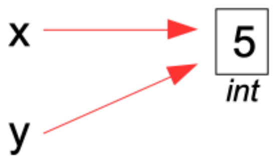
Creating Variables

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



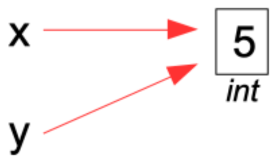
Creating Variables

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



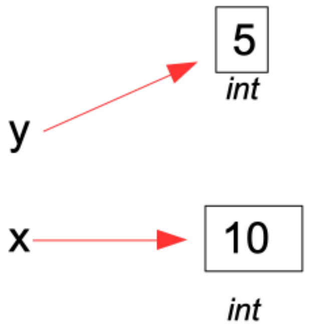
Creating Variables

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



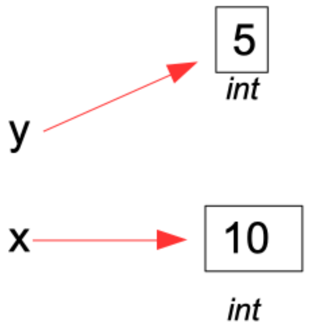
Creating Variables

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



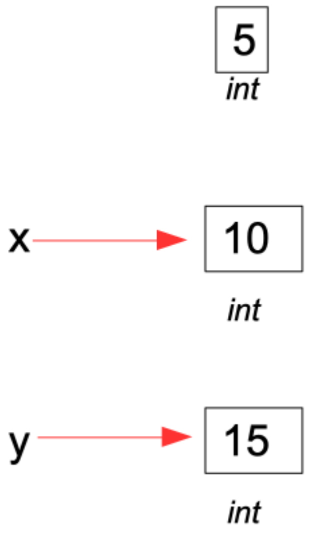
Creating Variables

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



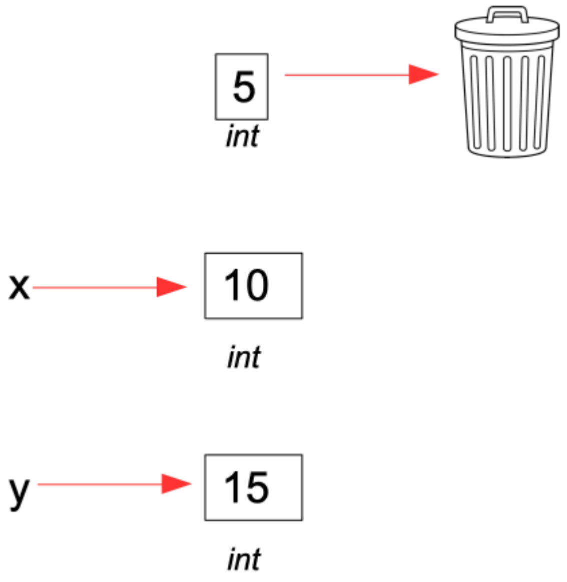
Creating Variables

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

f-string shortcut

Instead of

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

use

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

x is only typed once

Command line arguments

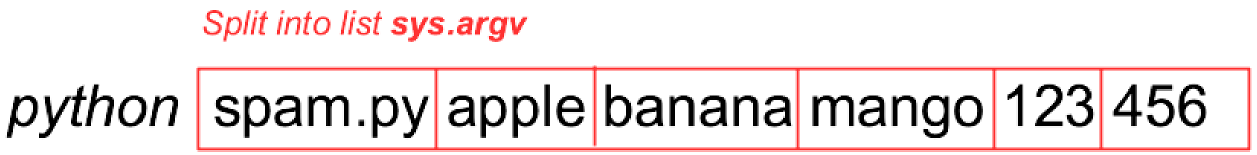
```
python spam.py apple banana mango 123 456
```

Command line arguments

All arguments to python interpreter

```
python spam.py apple banana mango 123 456
```

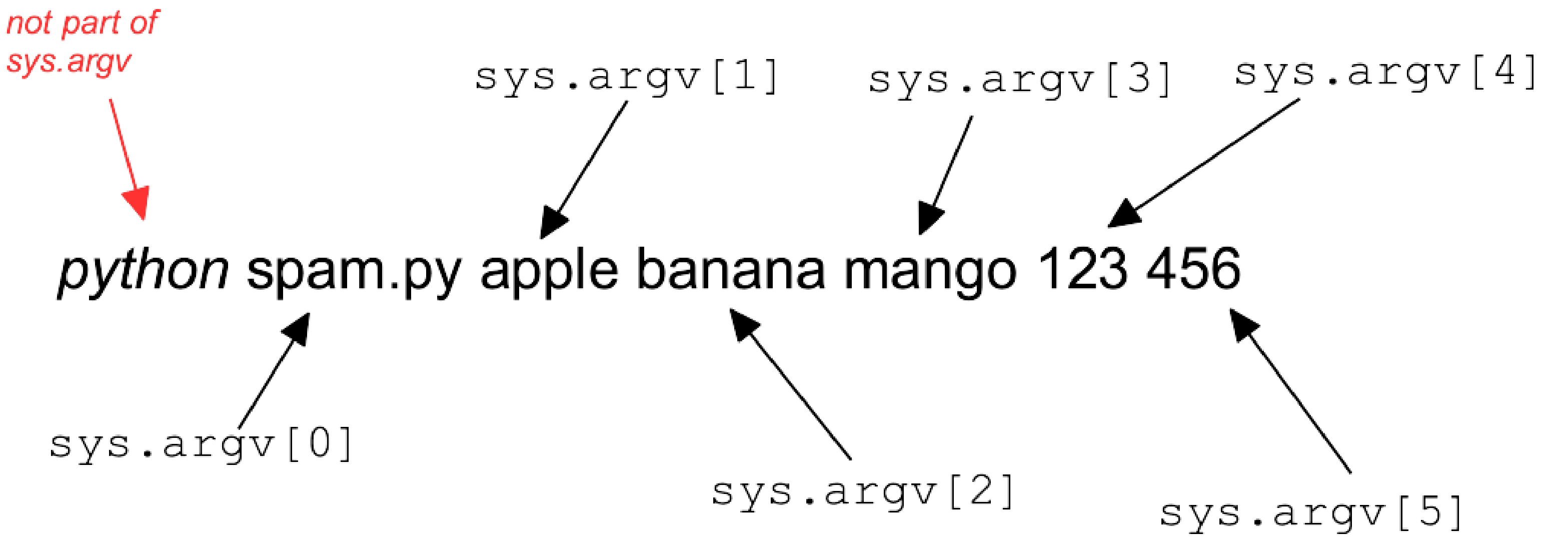
Command line arguments



Split into list ***sys.argv***

python spam.py apple banana mango 123 456

Command line arguments



not part of sys.argv

sys.argv[1] sys.argv[3] sys.argv[4]

python spam.py apple banana mango 123 456

sys.argv[0] sys.argv[2] sys.argv[5]

Indenting blocks

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

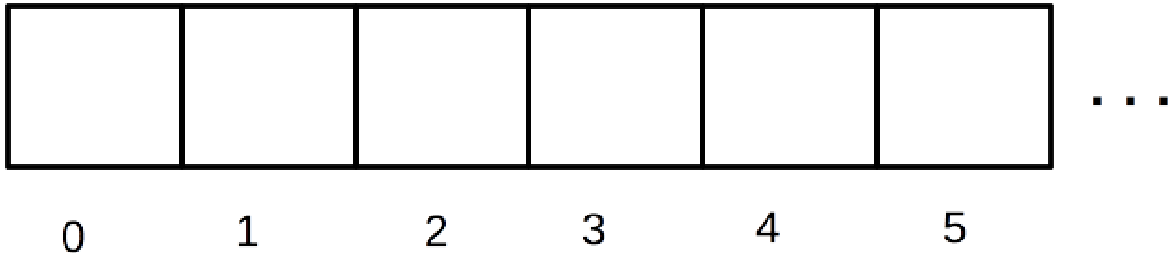
Boolean values

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

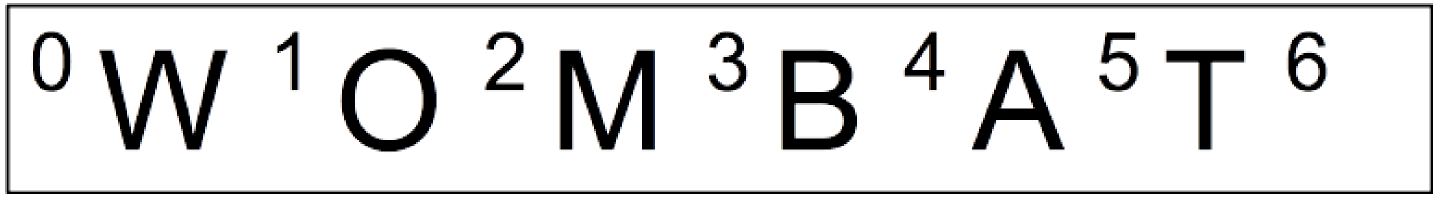
If X is	Boolean value of X is
`None`	`False`
`False`	`False`
`True`	`True`
anything else	`True`

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
Use case: iterating	Use case: indexing or unpacking
"ARRAY"	"STRUCT" or "RECORD"

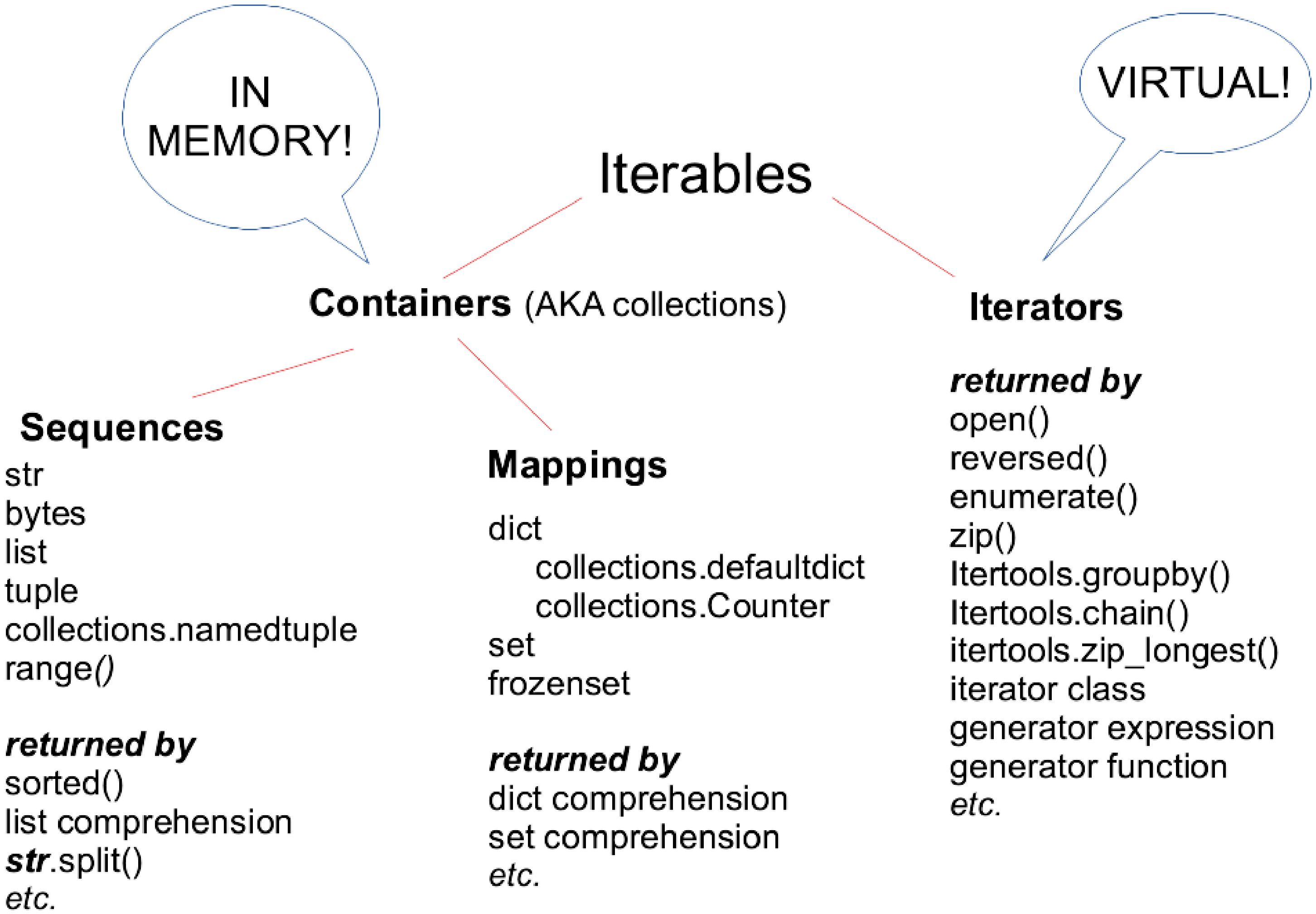
A Myth

Tuples are just read-only lists

Tuple alternatives

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

Iterables



IN MEMORY!

VIRTUAL!

Iterables

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`

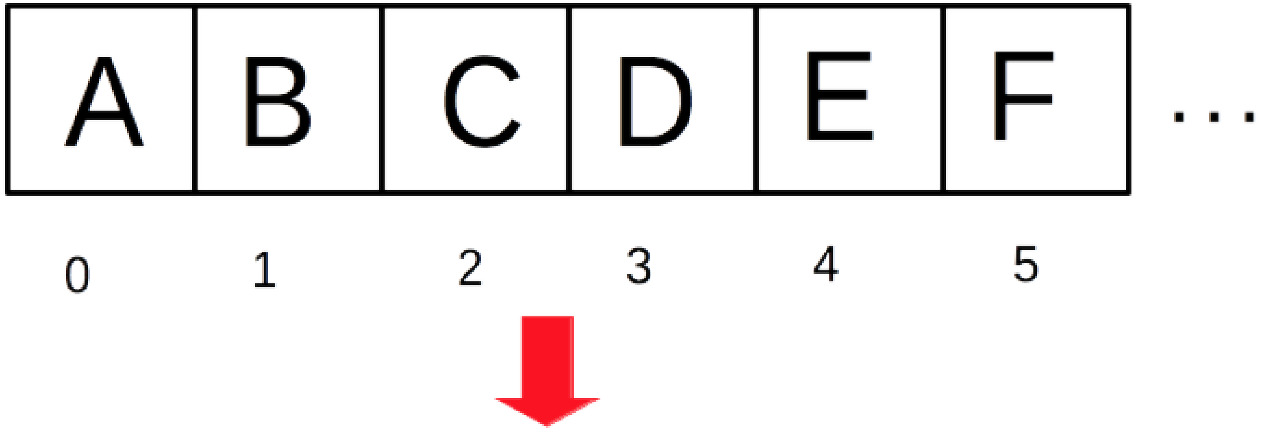
Iterators

- 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)...

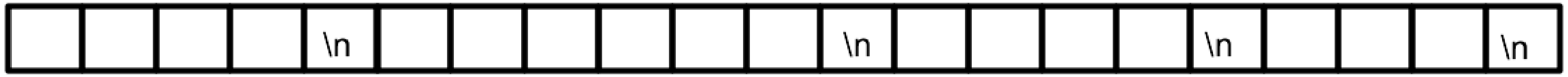
Using enumerate()

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

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

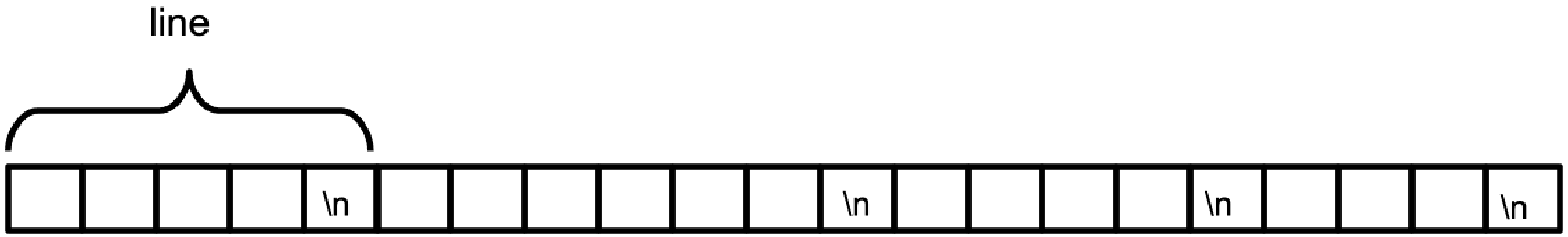
```
enumerate(letters, 1)
(1, 'alpha'), (2, 'beta'), (3, 'gamma')
```

Reading Text Files

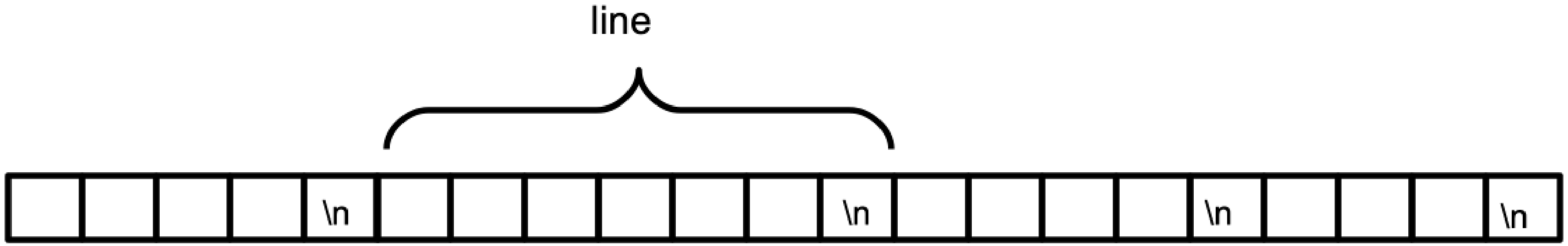


```
with open("somefile") as file_in:
```

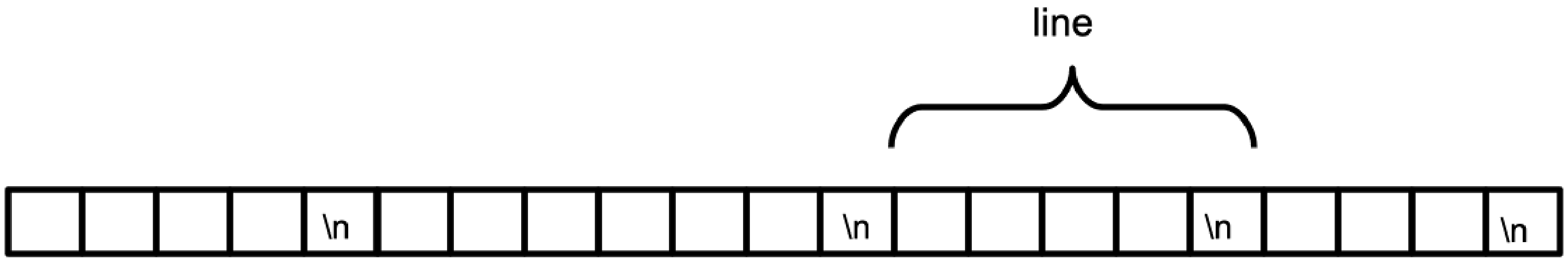
Reading one line at a time



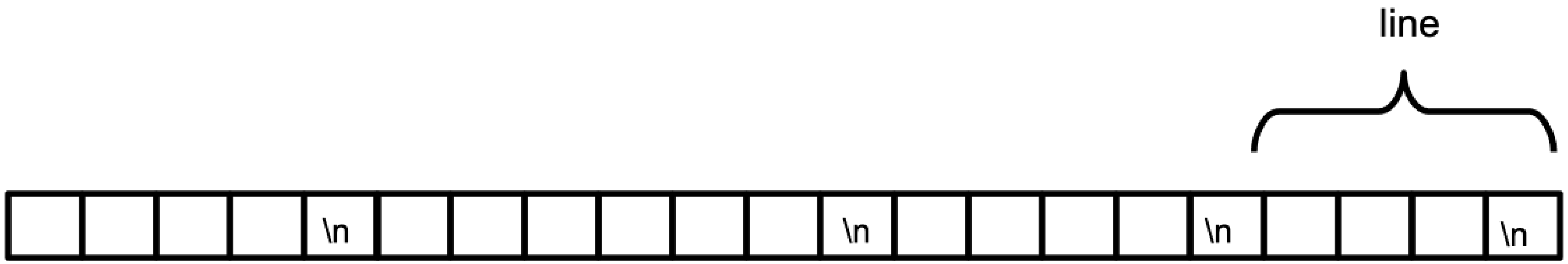
Reading one line at a time



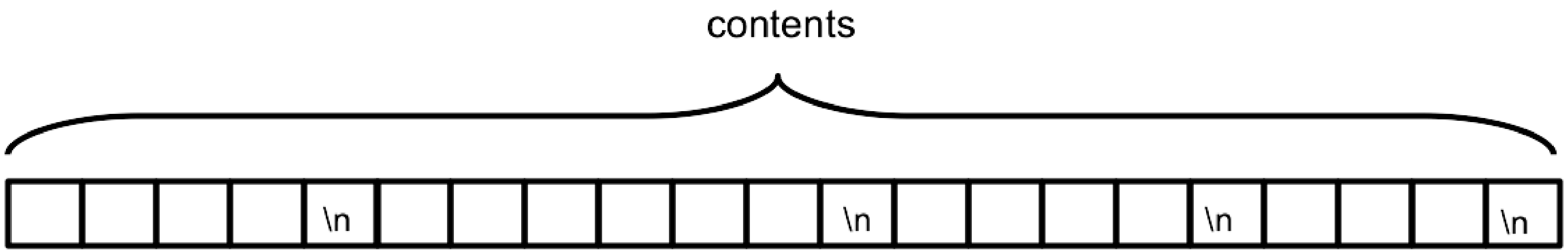
Reading one line at a time



Reading one line at a time

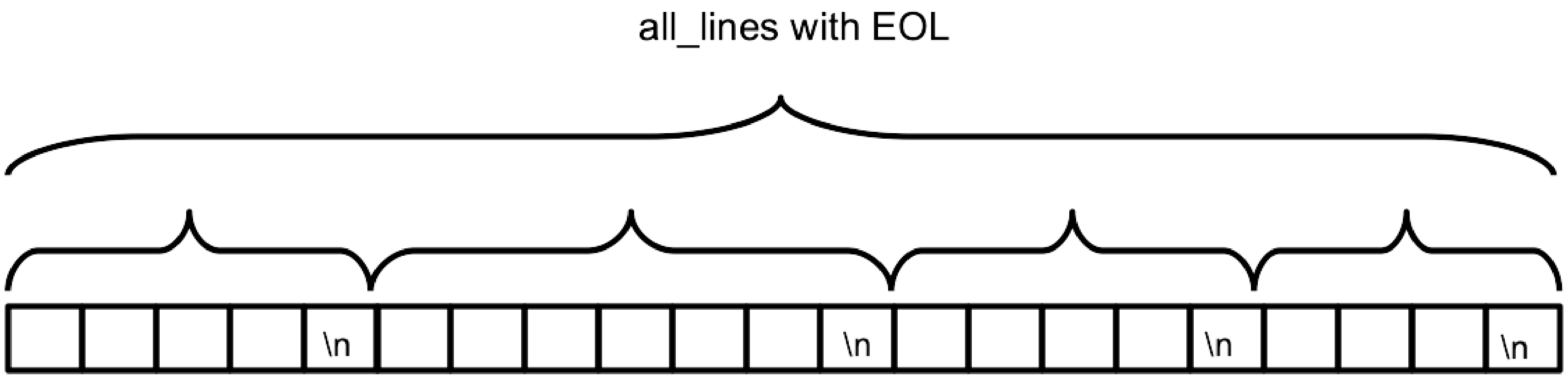


Reading entire file into string



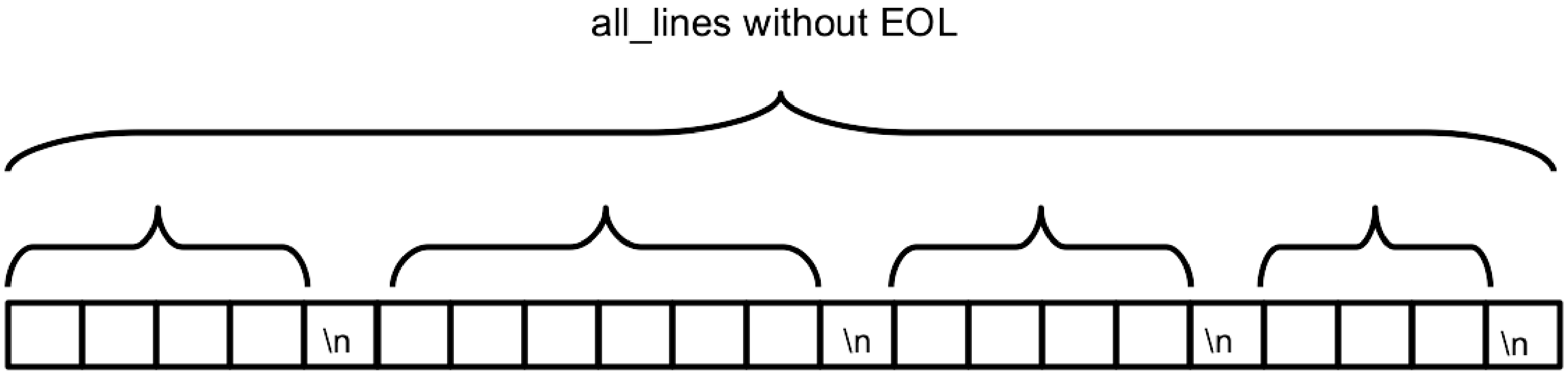
```
with open("somefile") as file_in:
    contents = file_in.read()
```

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()
```

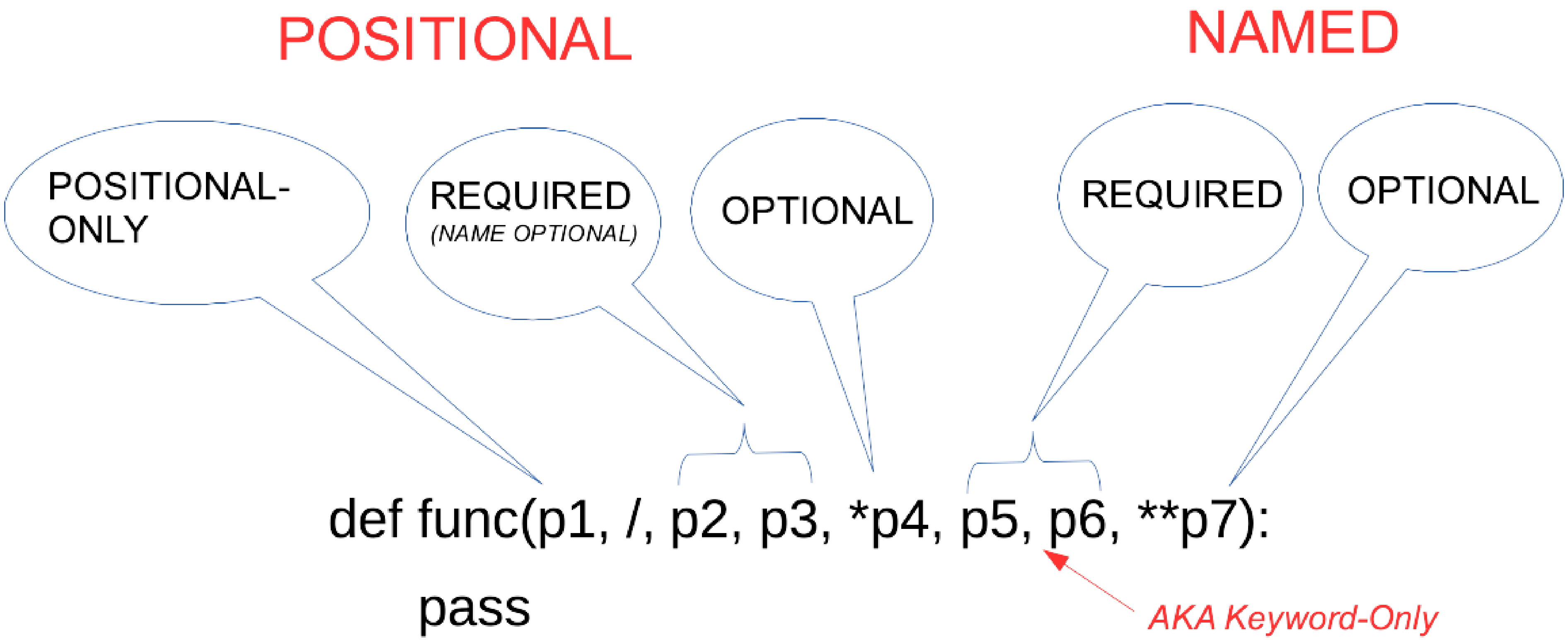
Dictionary

- 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



POSITIONAL

NAMED

POSITIONAL-ONLY

REQUIRED

(NAME OPTIONAL)

OPTIONAL

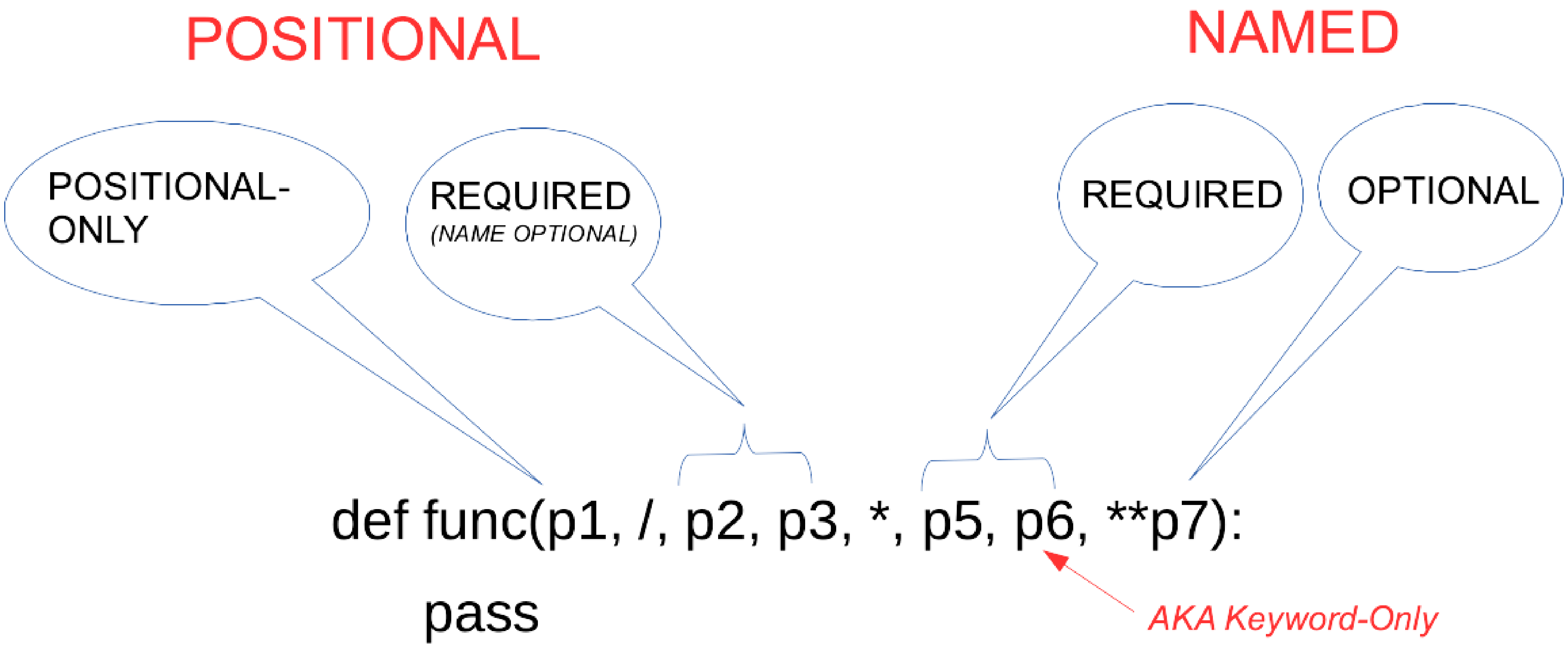
REQUIRED

OPTIONAL

```
def func(p1, /, p2, p3, *p4, p5, p6, **p7):
    pass
```

AKA Keyword-Only

Function parameters



POSITIONAL

NAMED

POSITIONAL-
ONLY

REQUIRED
(NAME OPTIONAL)

REQUIRED

OPTIONAL

```
def func(p1, /, p2, p3, *, p5, p6, **p7):
    pass
```

AKA Keyword-Only

Argument passing

~~Passing by reference~~

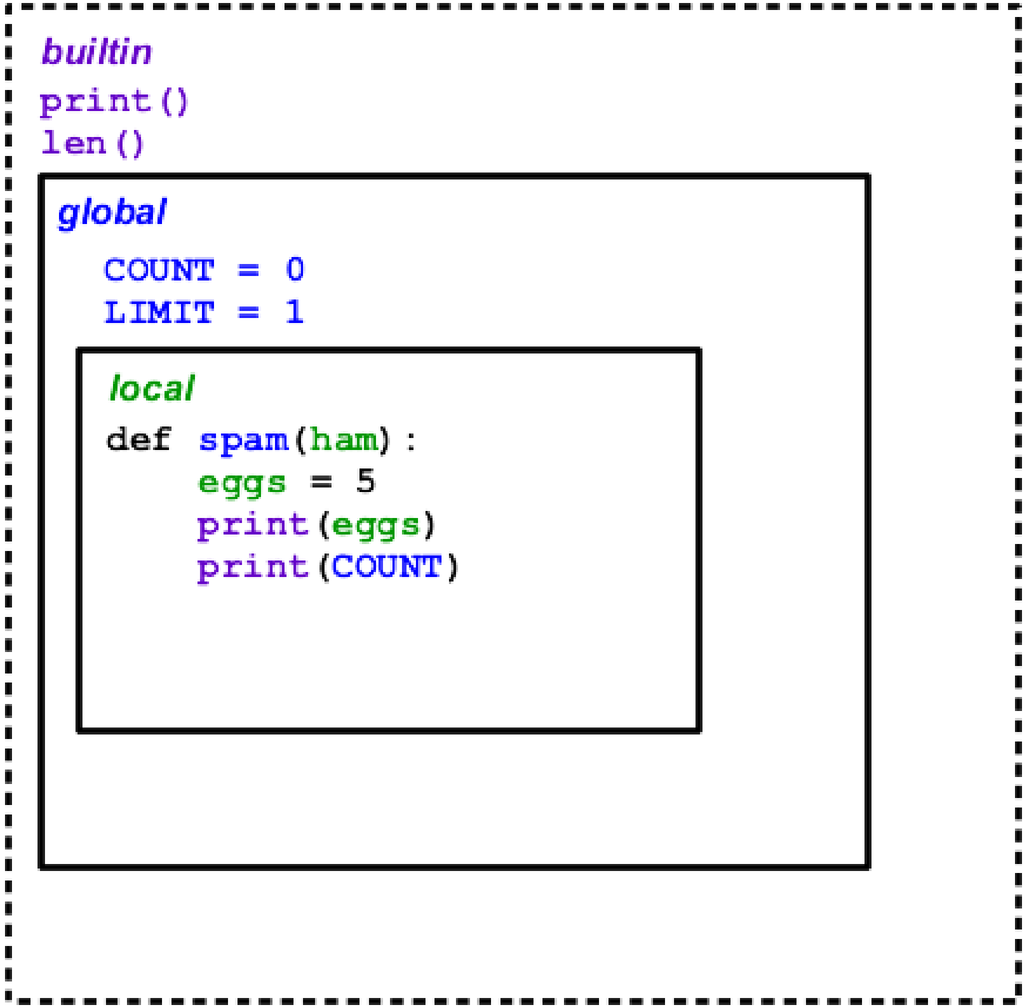
~~Passing by value~~

✓ 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



builtin

```
print()
len()
```

global

```
COUNT = 0
LIMIT = 1
```

local

```
def spam(ham):
    eggs = 5
    print(eggs)
    print(COUNT)
```

Variable scope

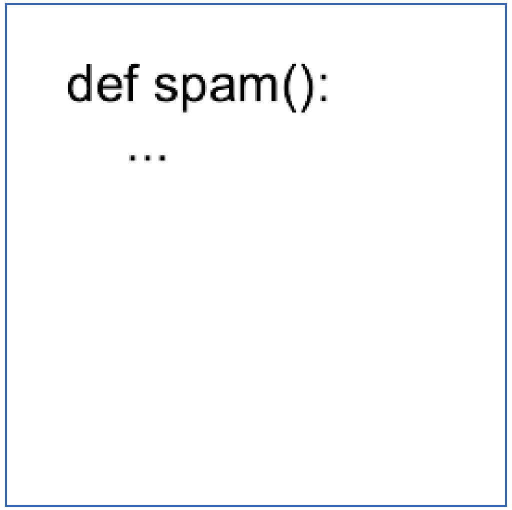
```python
ALPHA = 10

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

spam(1234)
```

BUILTIN

GLOBAL

LOCAL

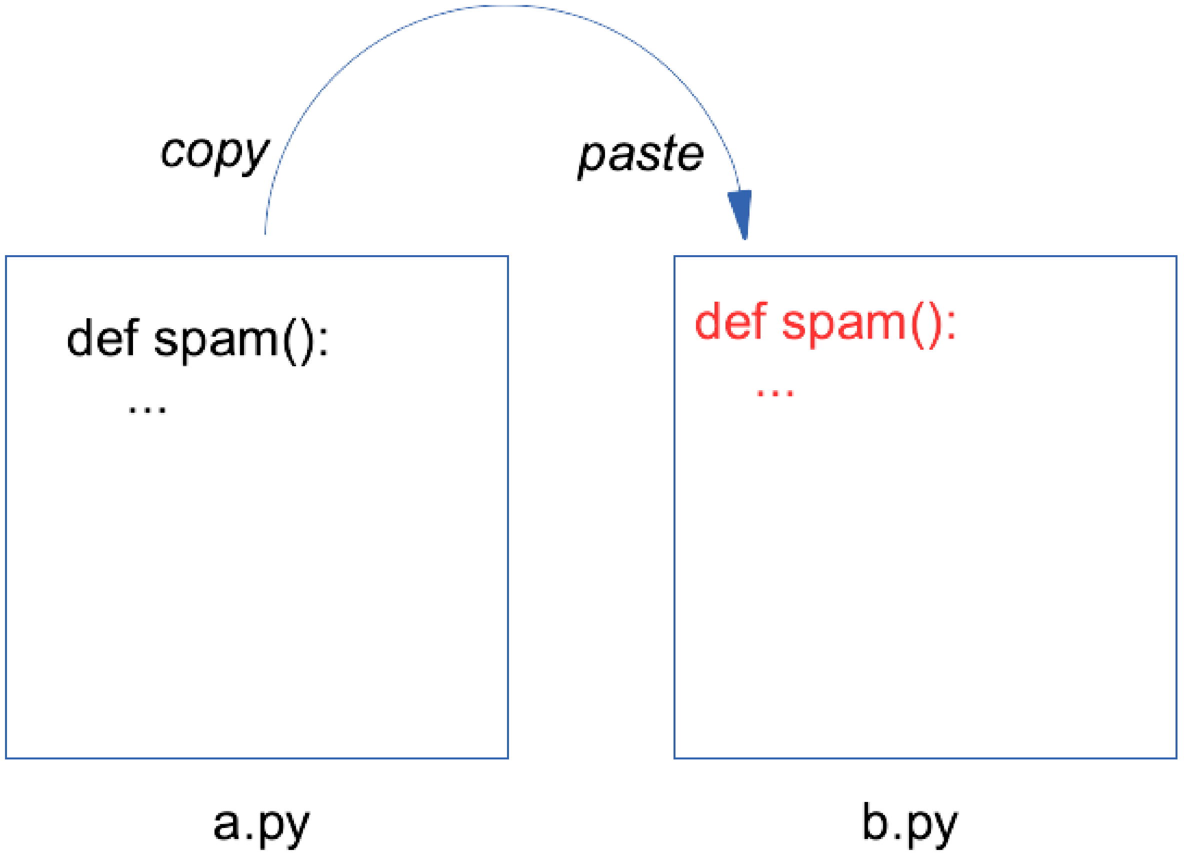
Copy/pasting functions



```
def spam():
    ...
```

a.py

Copy/pasting functions



copy

paste

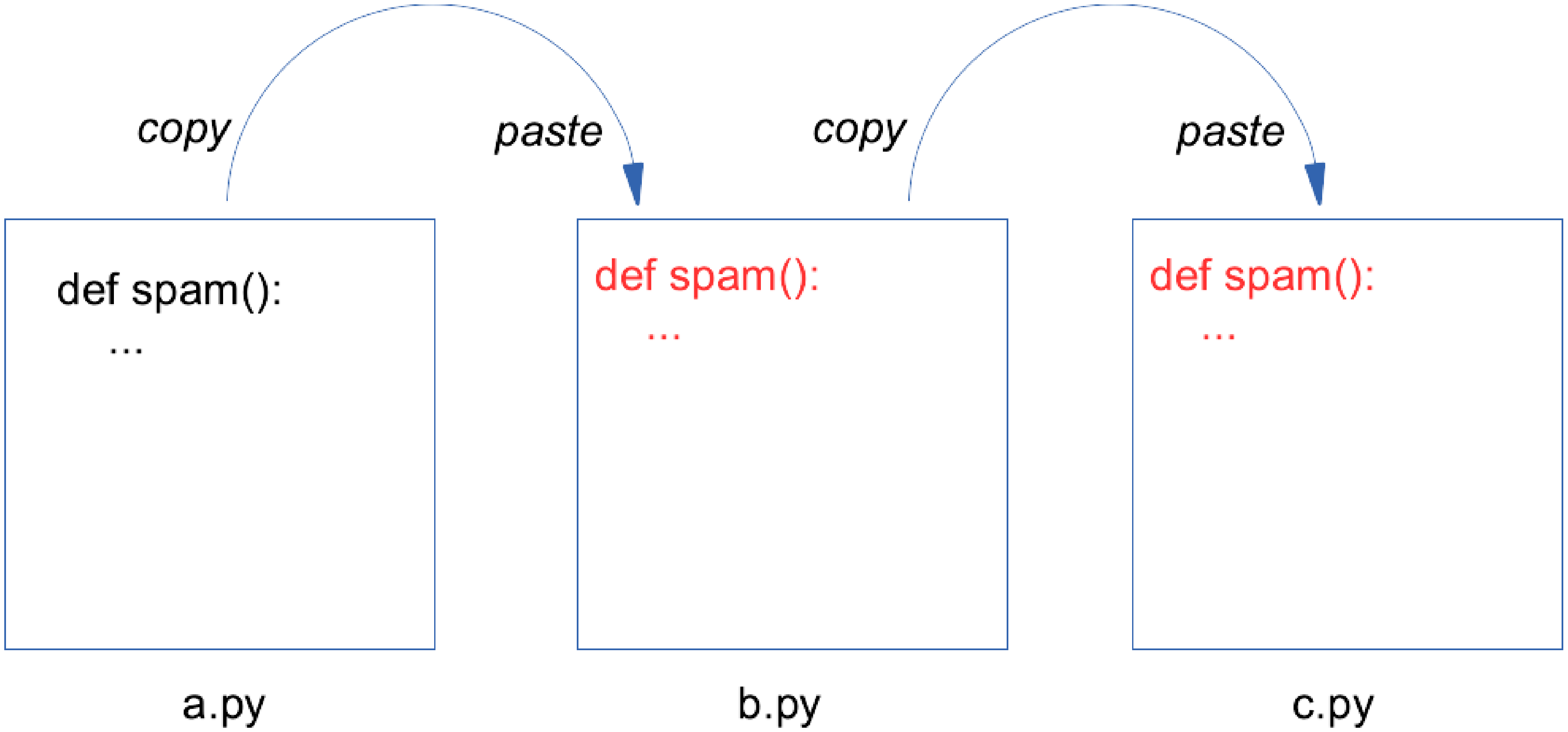
```
def spam():
    ...
```

a.py

```
def spam():
    ...
```

b.py

Copy/pasting functions



```
def spam():
    ...
```

a.py

```
def spam():
    ...
```

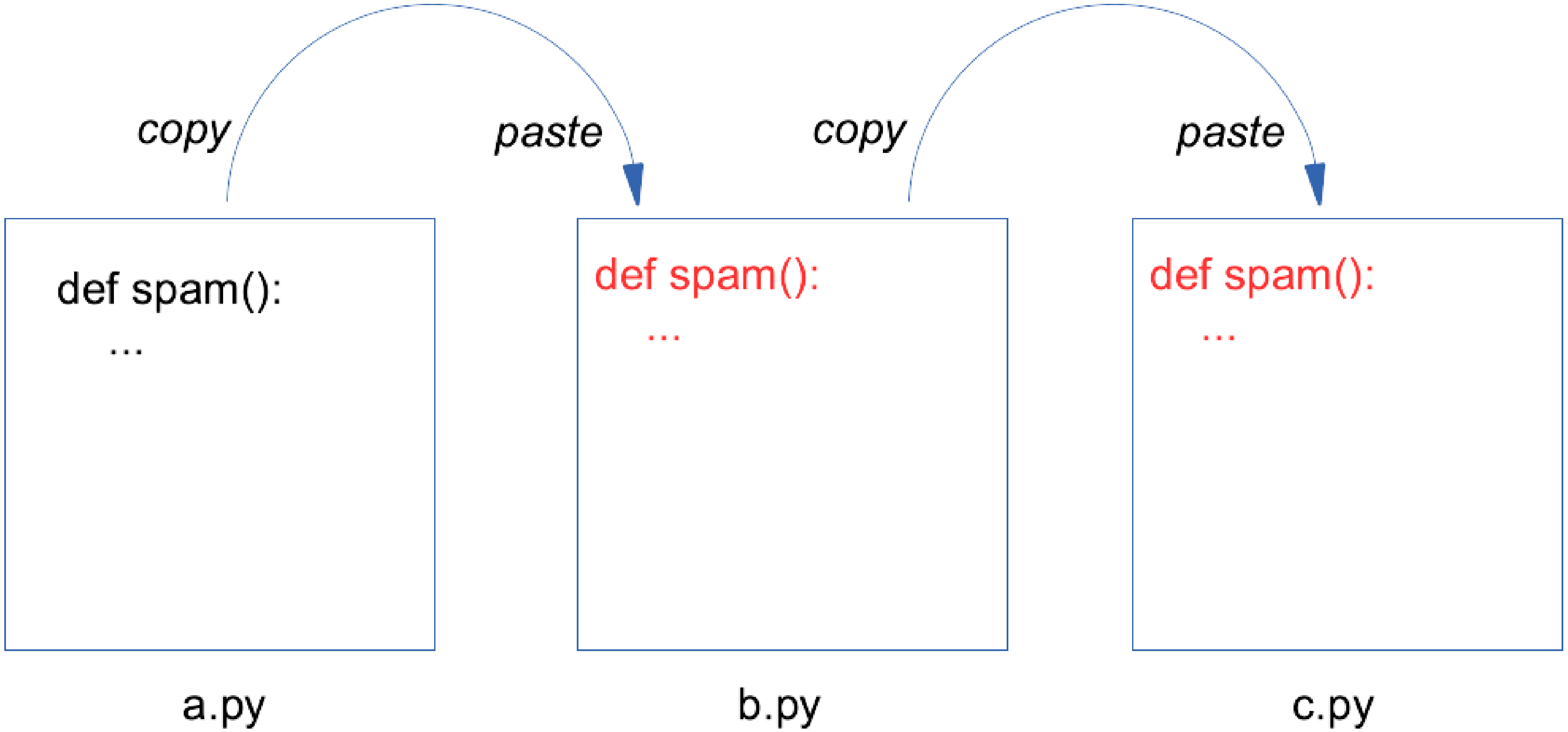
b.py

```
def spam():
    ...
```

c.py

Copy/pasting functions

DON'T DO THIS!!



```
def spam():
    ...
```

a.py

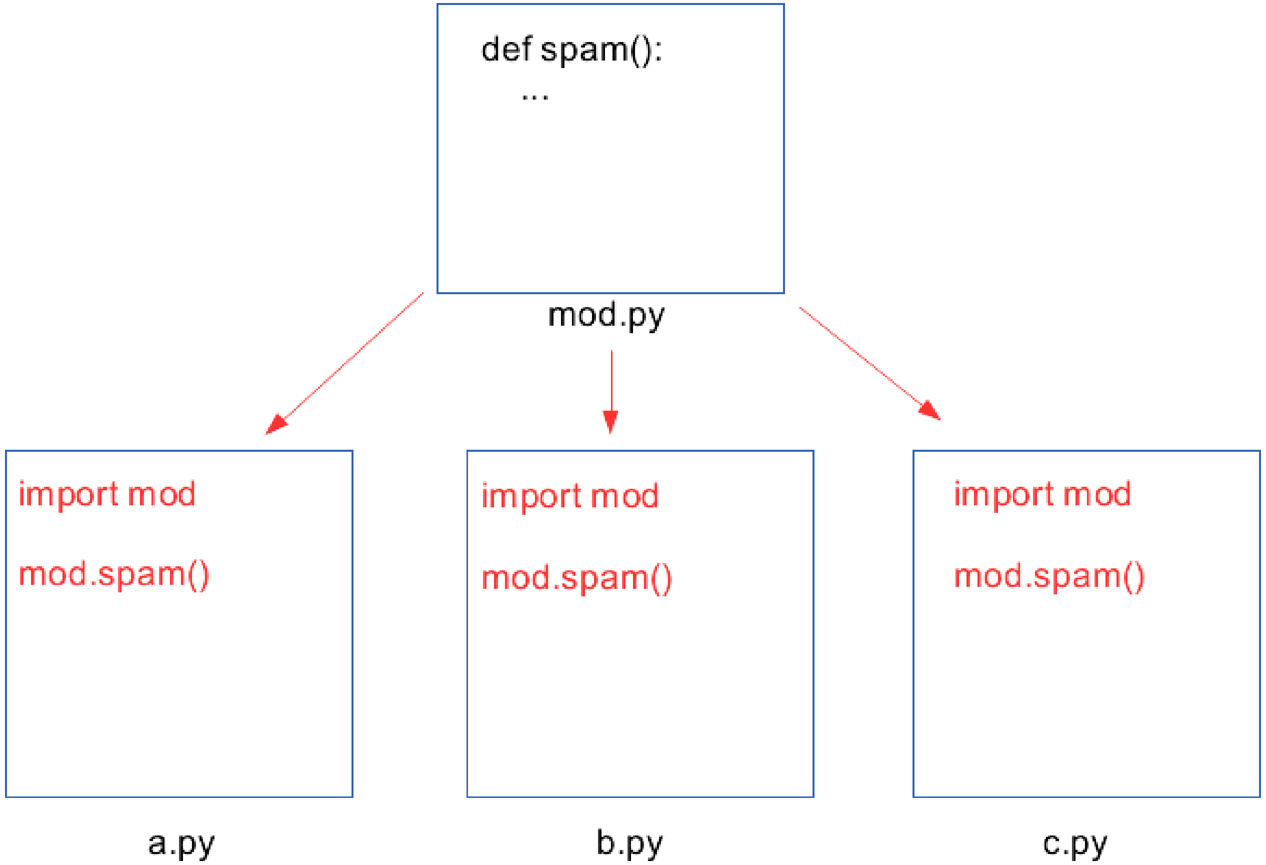
```
def spam():
    ...
```

b.py

```
def spam():
    ...
```

c.py

Using a module



```
def spam():
    ...
```

mod.py

```
import mod

mod.spam()
```

a.py

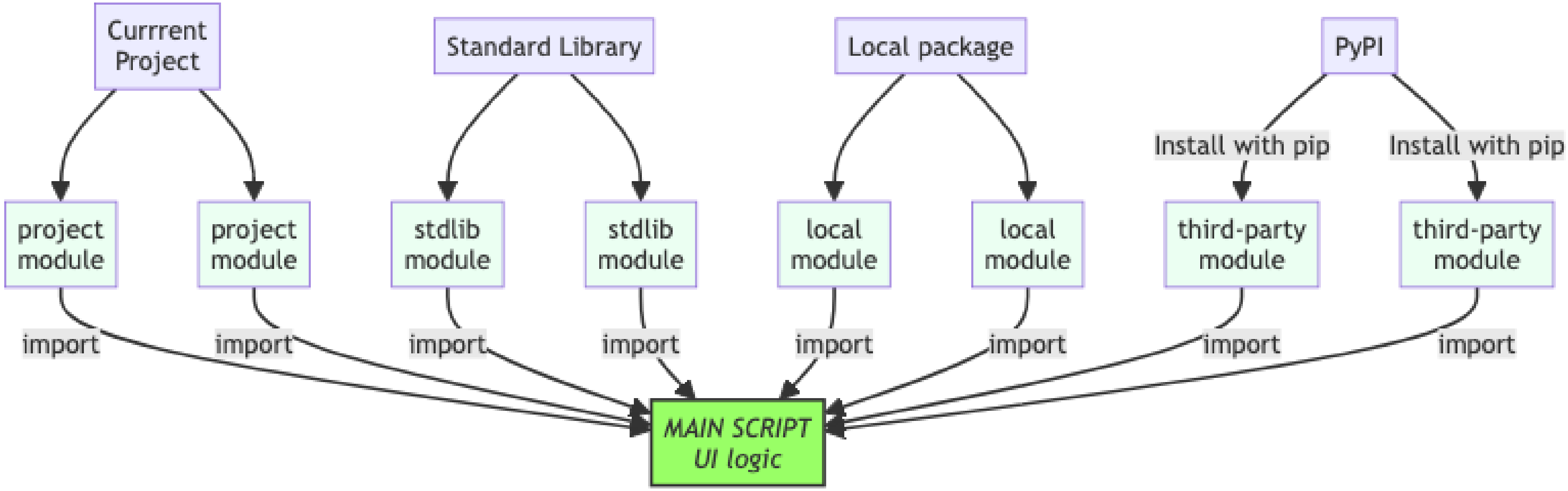
```
import mod

mod.spam()
```

b.py

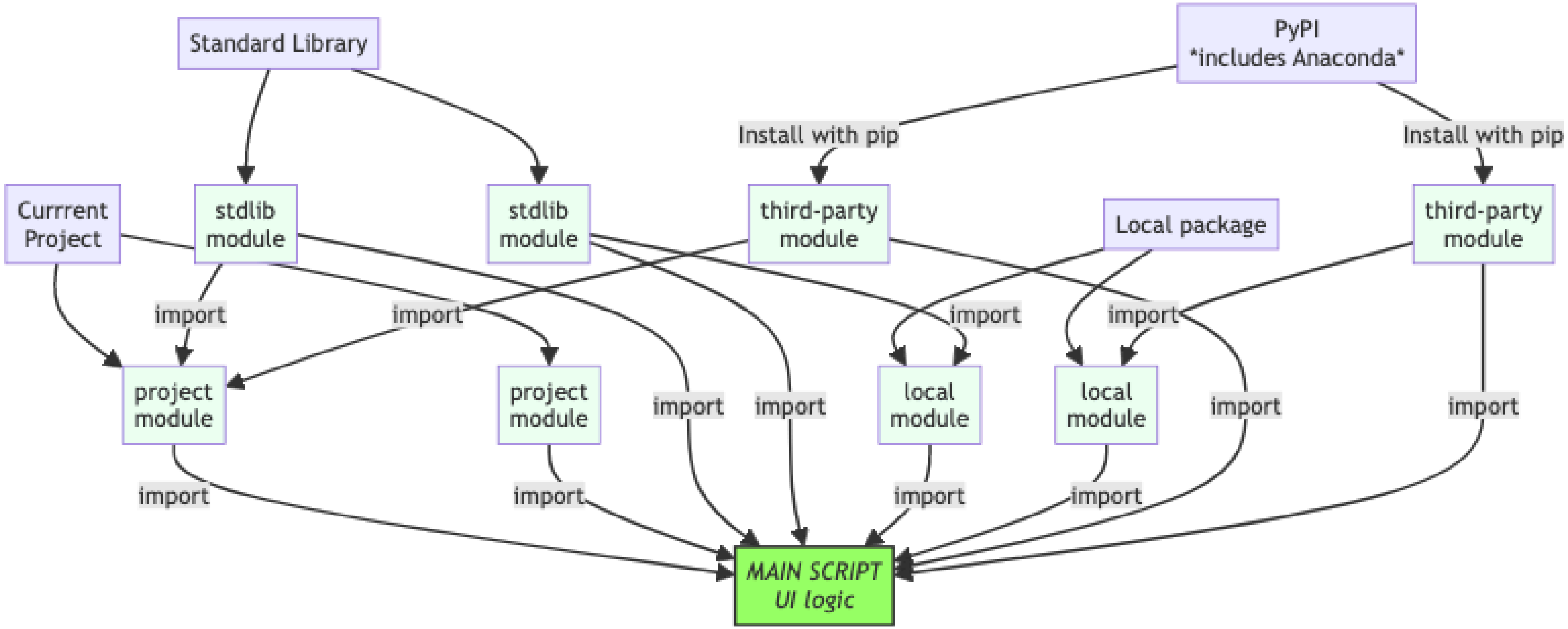
```
import mod

mod.spam()
```

c.py

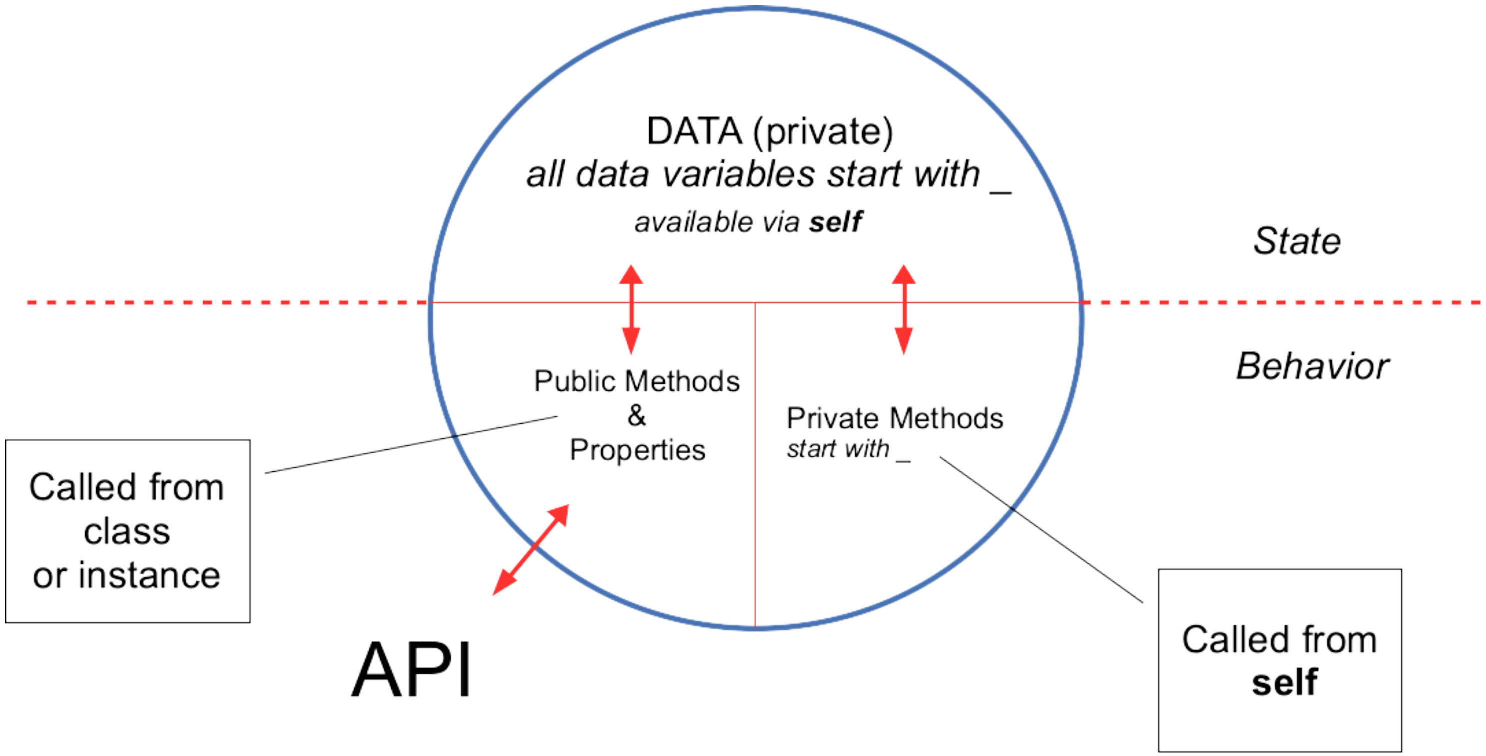
Project Imports



Project Imports (real-life)



A Python Class



try/except

```
try:
    # code that might have an exception
except (Exception1, Exception2):
    # code to handle Exception1 or Exception2
```

Multiple `except` blocks

```
try:
    # code that might have an exception
except (Exception1, Exception2):
    # code to handle Exception1 or Exception2
except Exception3:
    # code to handle Exception3
```

Using `else`

```
try:
    # code that might have an exception
except (Exception1, Exception2):
    # code to handle Exception1 or Exception2
except Exception3:
    # code to handle Exception3
else:
    # code that should run if there are no exceptions
```

Using `finally`

```
try:
    # code that might have an exception
except (Exception1, Exception2):
    # code to handle Exception1 or Exception2
except Exception3:
    # code to handle Exception3
else:
    # code that should run if there are no exceptions
finally:
    # code to remove any unneeded resources
```

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

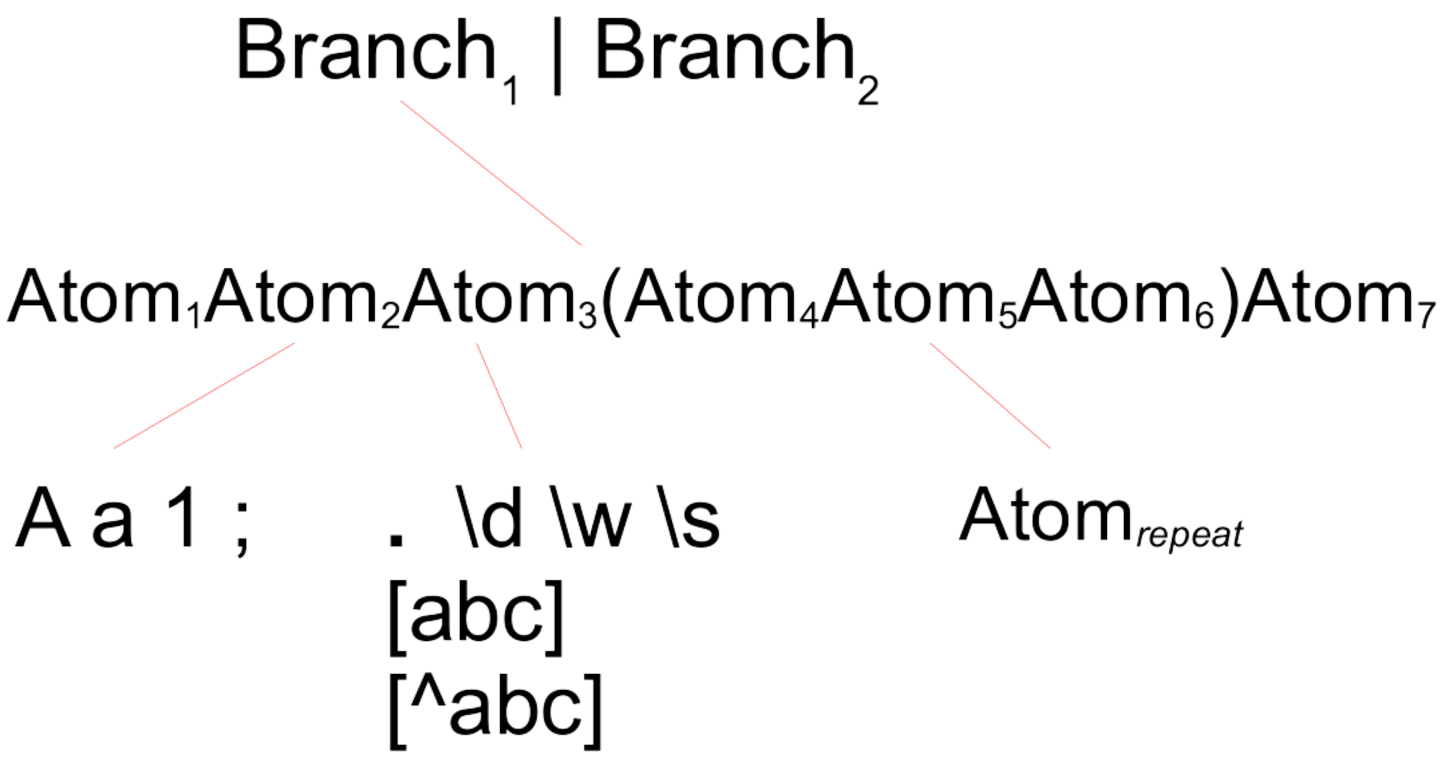
REPLACE

Substitute new text for match

SPLIT

Get what ***did not*** match

Regular Expression Components



Branch$_1$ | Branch$_2$

Atom$_1$Atom$_2$Atom$_3$(Atom$_4$Atom$_5$Atom$_6$)Atom$_7$

A a 1 ;

. \d \w \s

[abc]

[^abc]

Atom$_{repeat}$

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.

Sorting

Numbers

`n, n, n, …`

Strings

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

Nested iterables

[$\texttt{Obj}_1$, $\texttt{Obj}_2$, $\texttt{Obj}_3$], [$\texttt{Obj}_1$, $\texttt{Obj}_2$, $\texttt{Obj}_3$],

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

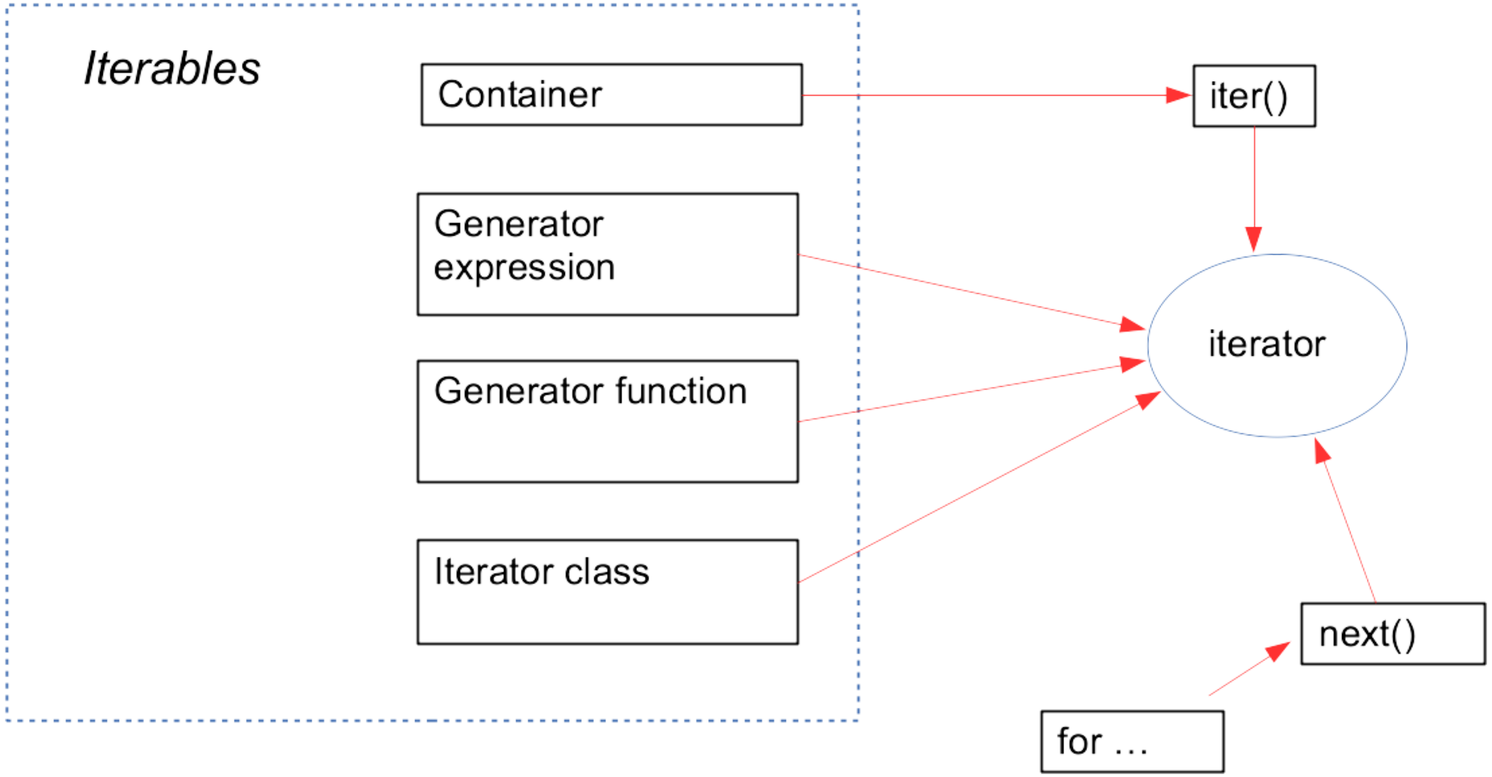
- dict comprehension

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

- set comprehension

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

Iterators


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

Distribution vs import package

Distribution package

what you install

Import package

what you import

Typical case

- Distribution package name: `fignewton`
- Import package name: `fignewton`

```
pip install fignewton
```

```
import fignewton
```

Alternate case

- Distribution package name: `python-fignewton`
- Import package name: `fignewton`

```
pip install python-fignewton
```

```
import fignewton
```

Real life examples

Distribution package (use with pip)	**Import package**
Pillow	`pil`
beautifulsoup4	`bs4`
PyYAML	`yaml`
python-magic	`magic`
crispy-bootstrap4	`crispy_bootstrap4`

Invoking Python

- Specify path to file or folder

```
python script.py
```

- Use `sys.path` (includes `PYTHONPATH`)

```
python -m module
```

- Runs all code, including code in

```
if __name__ == '__main__.py':
```

Specify path to file/folder

`python FILE`

Run all code in `FILE`

`python FOLDER/FILE`

Run all code in `FOLDER/FILE`

`python FOLDER`

Run all code in `FOLDER/__main__.py`

Find via sys.path

```
python -m MODULE
```

Run all code in `MODULE`

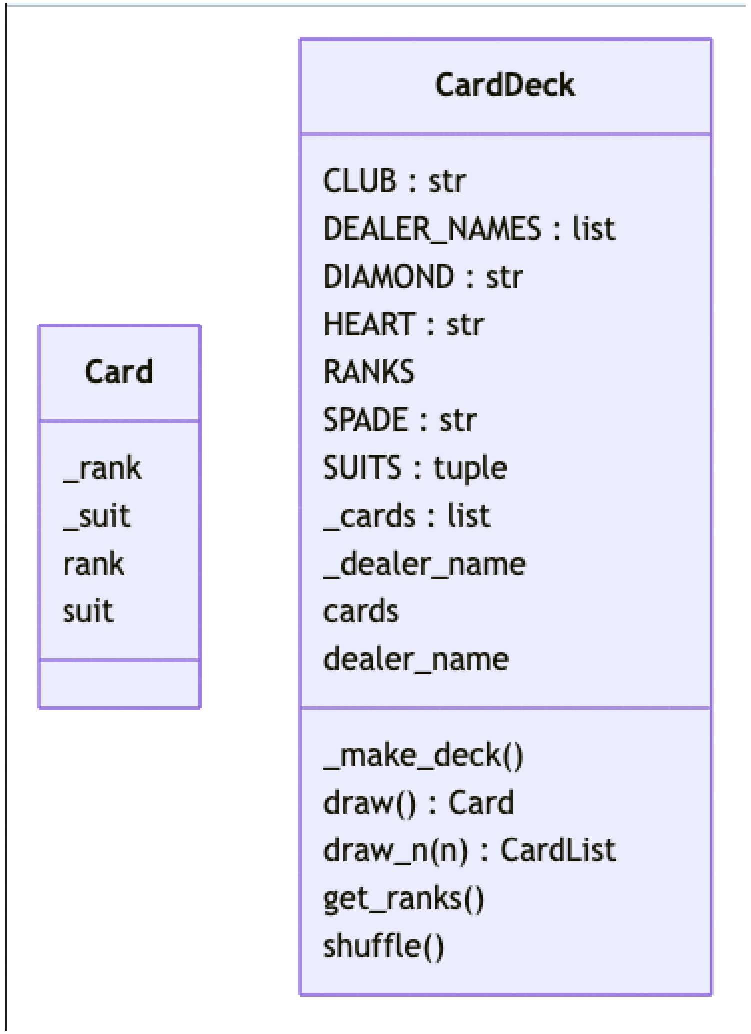
```
python -m PACKAGE
```

Run all code in `PACKAGE.__init__.py`
Run all code in `PACKAGE.__main__.py`

```
python -m PACKAGE.MODULE
```

Run all code in `PACKAGE.__init__.py`
Run all code in `PACKAGE.MODULE`

pyreverse (classes)



Card

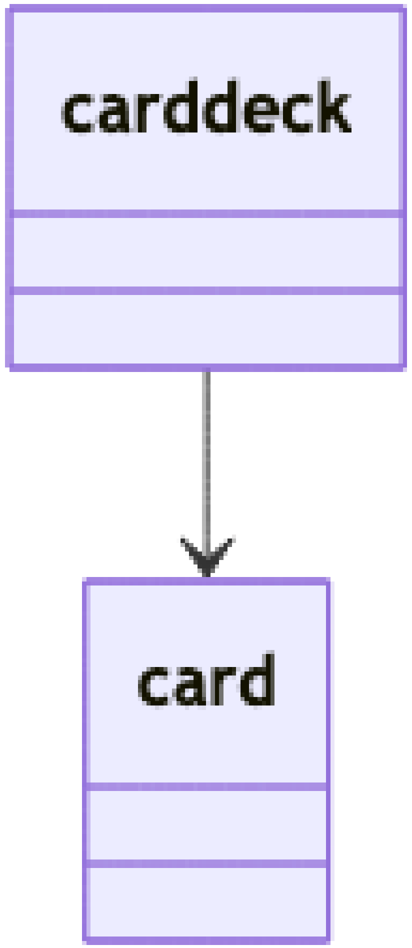
_rank
_suit
rank
suit

CardDeck

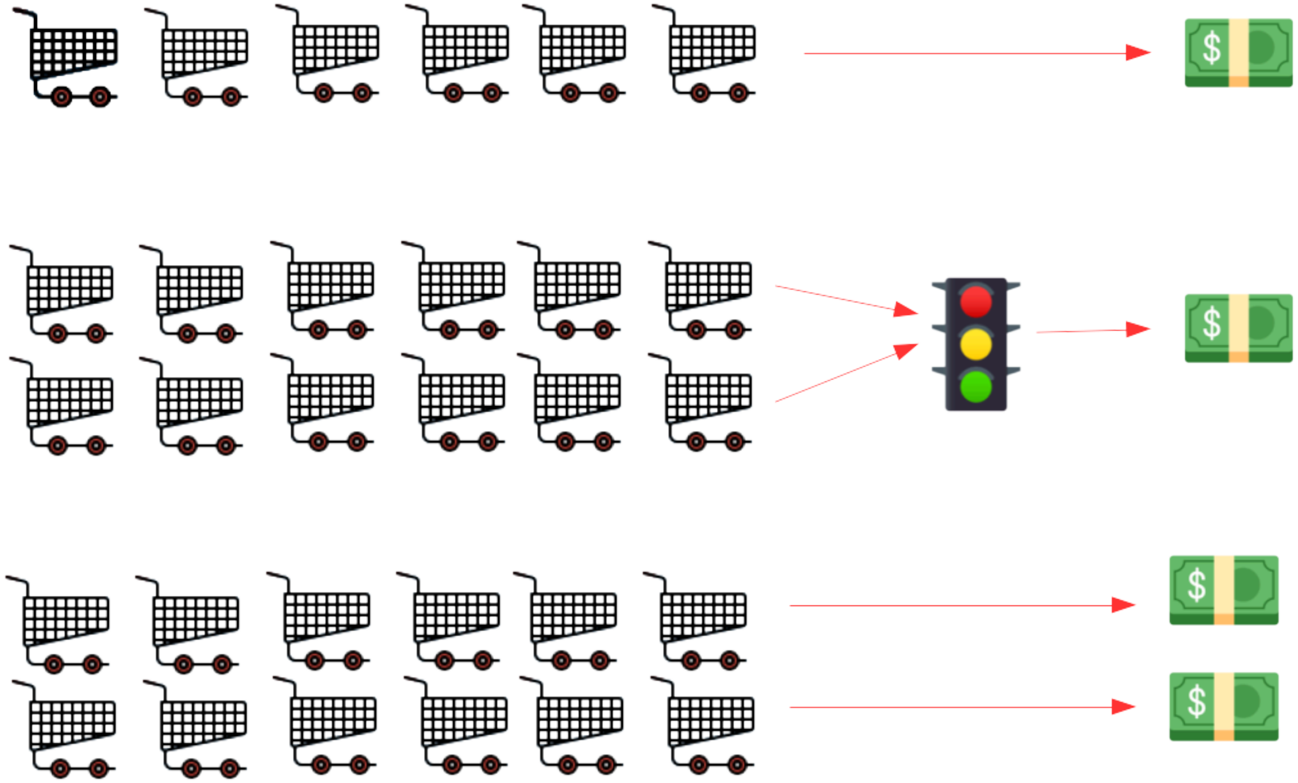
CLUB : str
DEALER_NAMES : list
DIAMOND : str
HEART : str
RANKS
SPADE : str
SUITS : tuple
_cards : list
_dealer_name
cards
dealer_name

_make_deck()
draw() : Card
draw_n(n) : CardList
get_ranks()
shuffle()

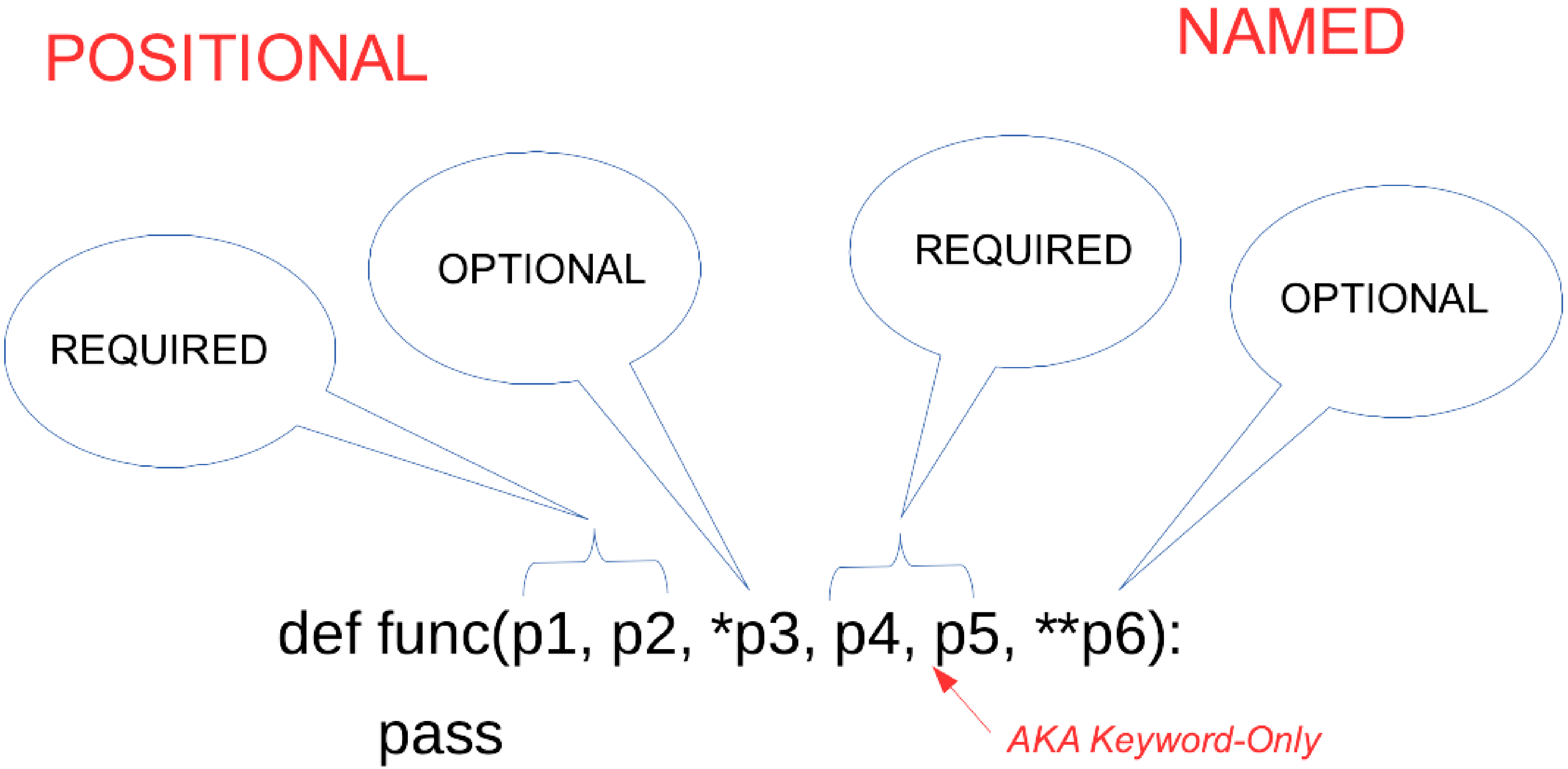
pyreverse (packages)



Concurrency



Function parameters



POSITIONAL

NAMED

REQUIRED

OPTIONAL

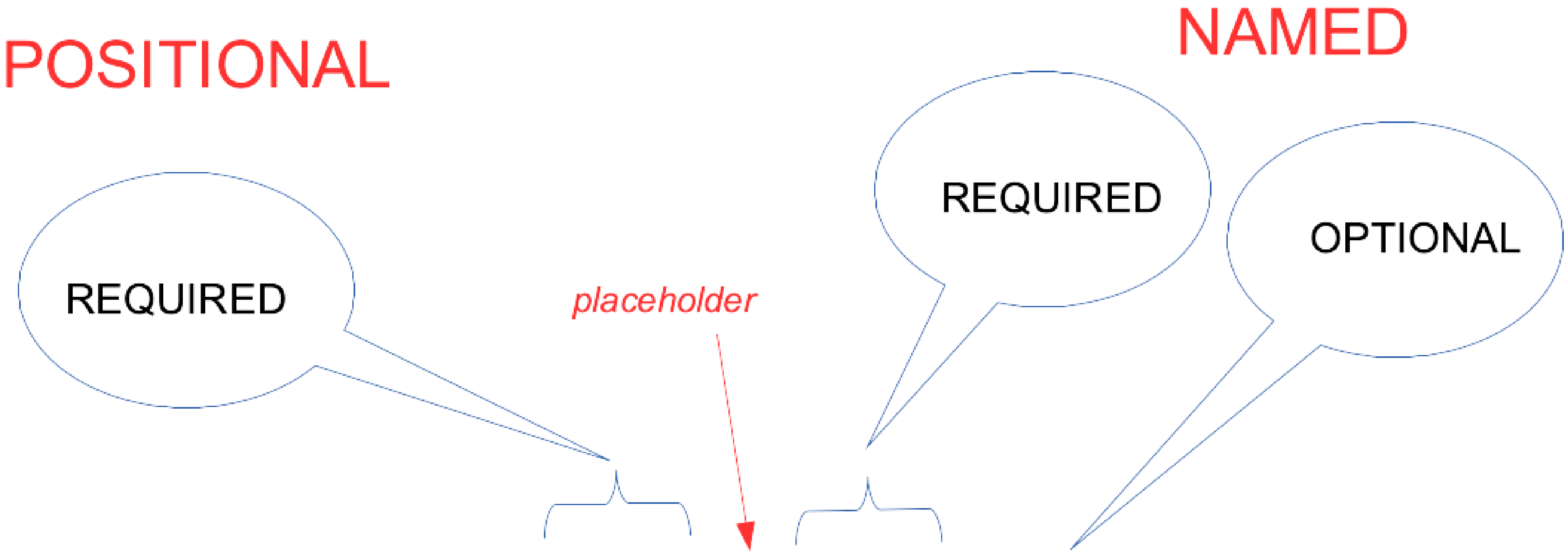
REQUIRED

OPTIONAL

```
def func(p1, p2, *p3, p4, p5, **p6):
    pass
```

AKA Keyword-Only

Function parameters



POSITIONAL

REQUIRED

placeholder

NAMED

REQUIRED

OPTIONAL

```
def func(p1, p2, *, p4, p5, **p6):
    pass
```

Pandas Location Accessors

Consistent access to rows, columns, and values

Select by *NAME* (column or index as string or number)

`.loc[ ]`

`.at[ ]`

Select by *POSITION* (column or row as integer)

`.iloc[ ]`

`.iat[ ]`

.loc[ ]

- Index/Column selector
  - single name `"spam"`
  - iterable of names `["spam", "ham", "eggs"]`
  - range of names `["spam":"toast"]`
  - boolean test/query `df_cust['state'] == 'VA'` *(rows only)*

```
df.loc[index-spec]                row(s) + all columns
df.loc[index-spec, column-spec]   rows(s) + column(s)
df.loc[:, column-spec]            all rows + column(s)
df.loc[:, df['col'] > 5]          all rows + column(s) with values > 5
```

.iloc[ ]

- position specification
  - single position 5
  - iterable of positions `[5, 6, 7, 8]`
  - range of positions `[5:8]`

```
df.iloc[row-spec]                 row(s) + all columns
df.iloc[row-spec, column-spec]    rows(s) + column(s)
df.iloc[:, column-spec]           all rows + column(s)
```

.at[ ] & .iat[ ]

```
df.at[row-name, column-name]                value at row and column names

df.iat[row-position, column-position] value at row and column positions
```