

# Introduction to Python for JPMC

TTPS4802-GKJ

# 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

# 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

# Creating Variables

```
x = 5
```

# Creating Variables

`x = 5`



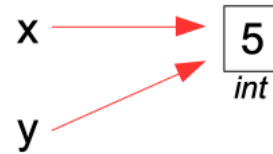
# Creating Variables

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



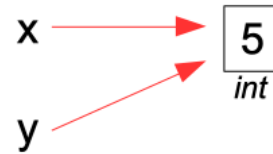
# Creating Variables

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



# Creating Variables

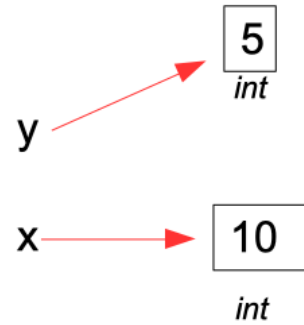
```
x = 5  
y = x  
x = 10
```





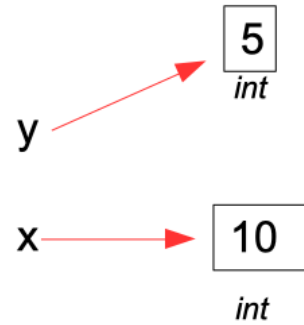
# Creating Variables

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



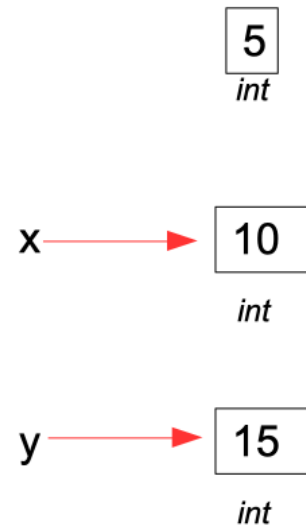
# Creating Variables

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



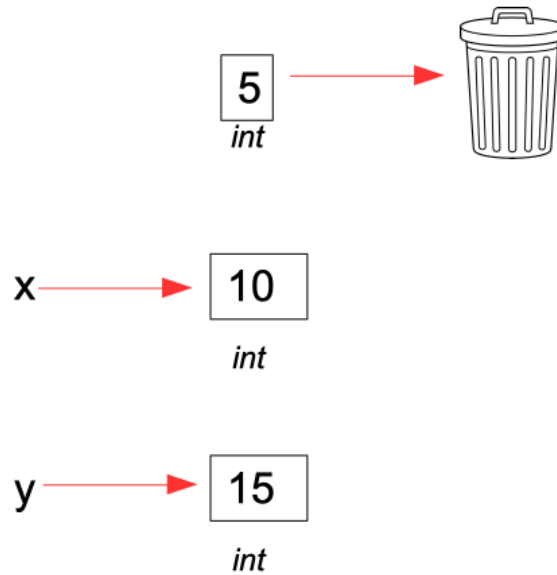
# Creating Variables

```
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()

<b>str()</b>	<b>repr()</b>
For humans	How to <b>re</b> produce 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

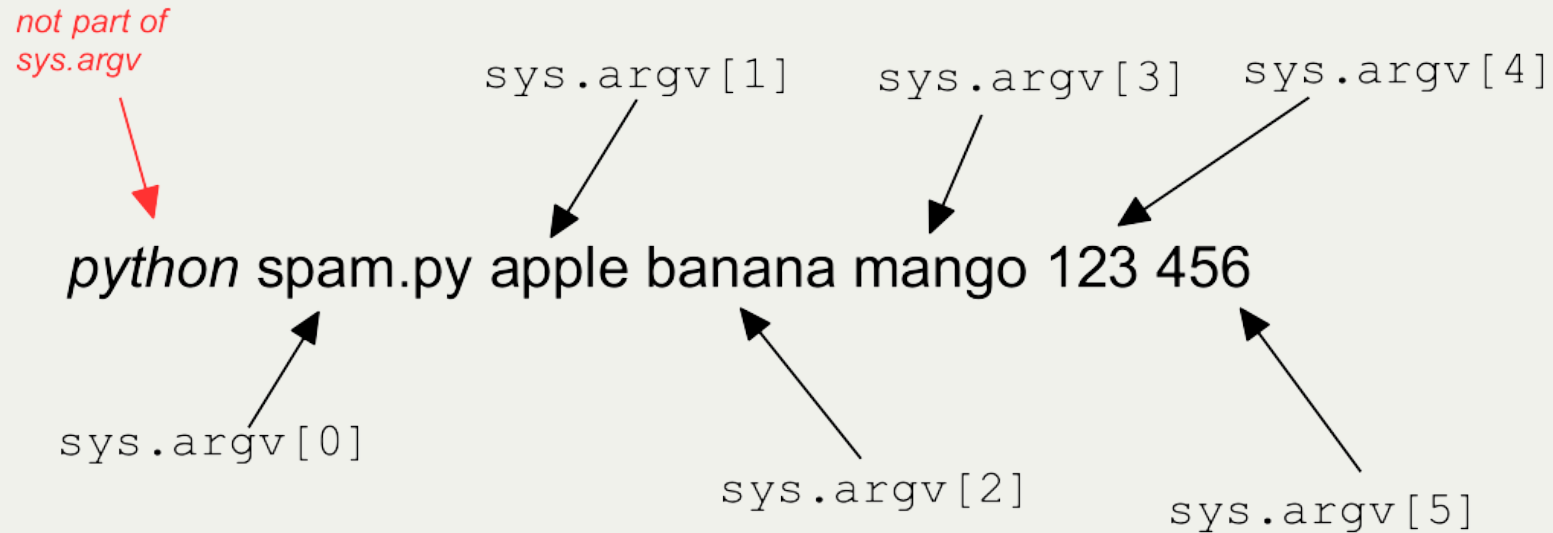
# Command line arguments

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

*python*

spam.py	apple	banana	mango	123	456
---------	-------	--------	-------	-----	-----

# Command line arguments



# 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

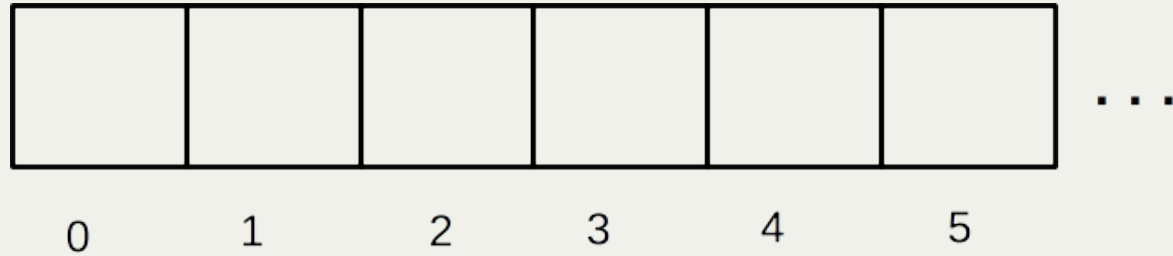
If X is	Boolean value of X is
Numeric, and equal to 0	False
Numeric, and NOT equal to 0	True
A collection, and len(X) is 0	False
A collection, and len(X) is > 0	True



# Boolean values

<b>If X is</b>	<b>Boolean value of X is</b>
None	False
False	False
True	True
<i>anything else</i>	True

# Sequences



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

# Slices

0	W	1	O	2	M	3	B	4	A	5	T	6
---	---	---	---	---	---	---	---	---	---	---	---	---

```
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

<b>Lists</b>	<b>Tuples</b>
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"

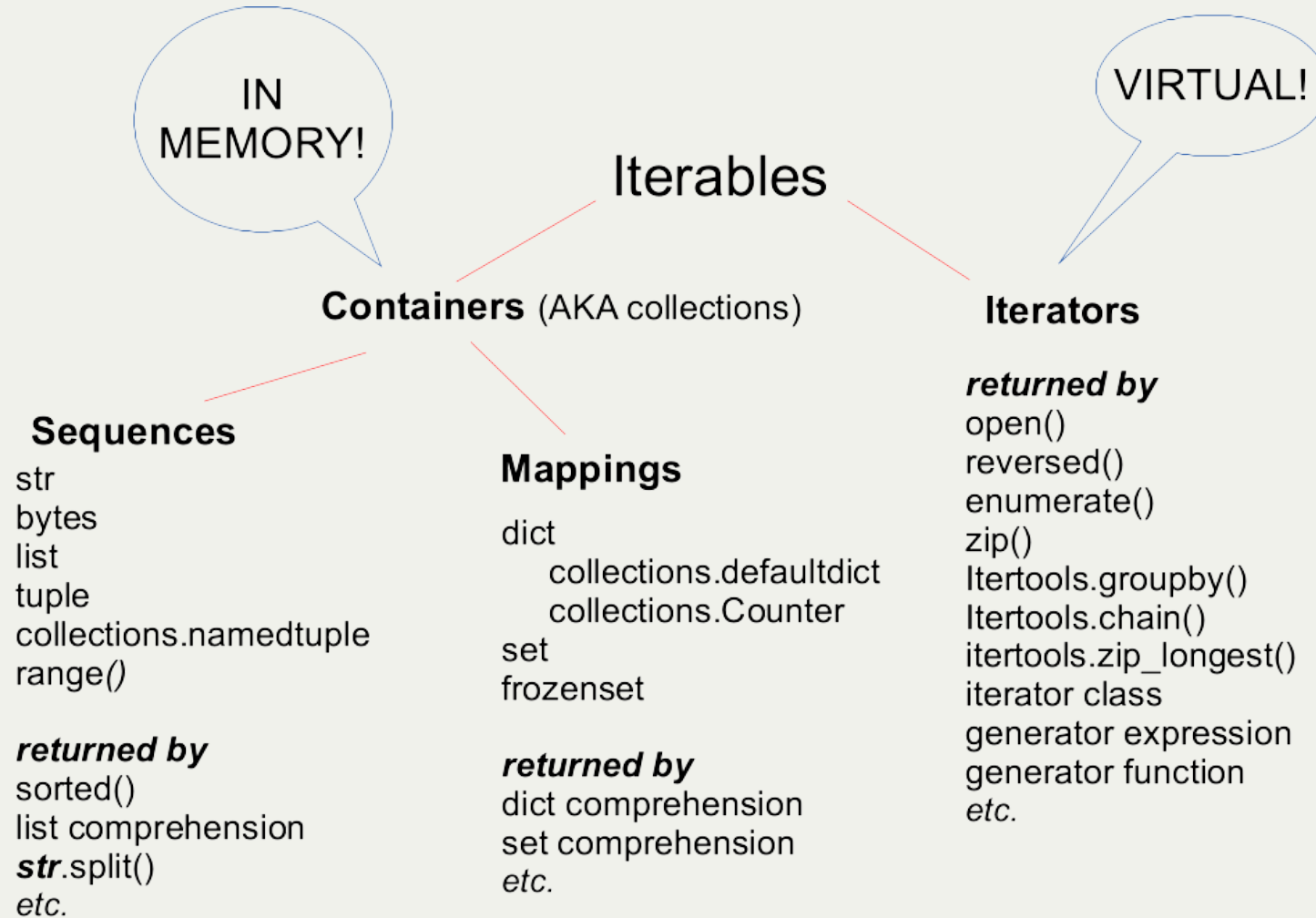
# A Myth

Tuples are just read-only lists

# Tuple alternatives

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

# Iterables



# 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`

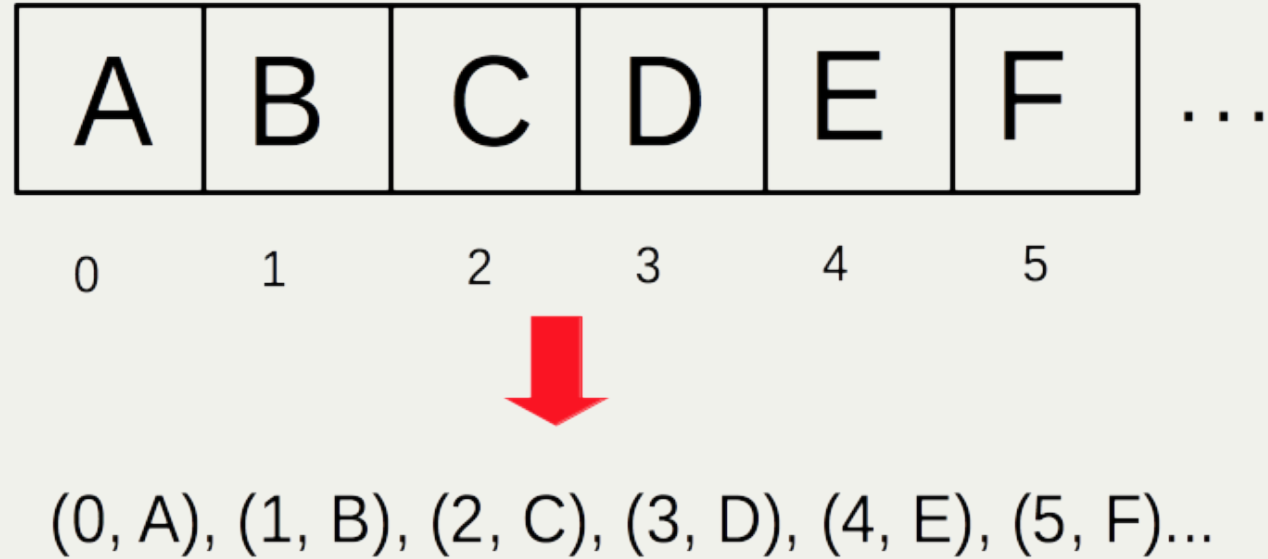
# Iterators

- 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



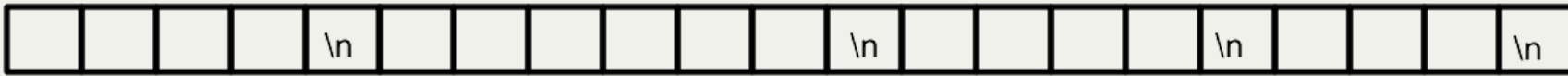
# Using enumerate()

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

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

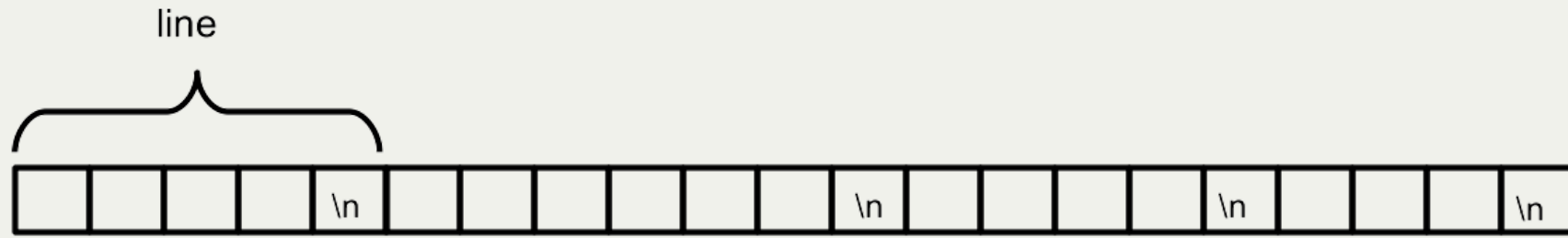
```
enumerate(letters, 1)  
(1, 'alpha'), (2, 'beta'), (3, 'gamma')
```

# Reading Text Files



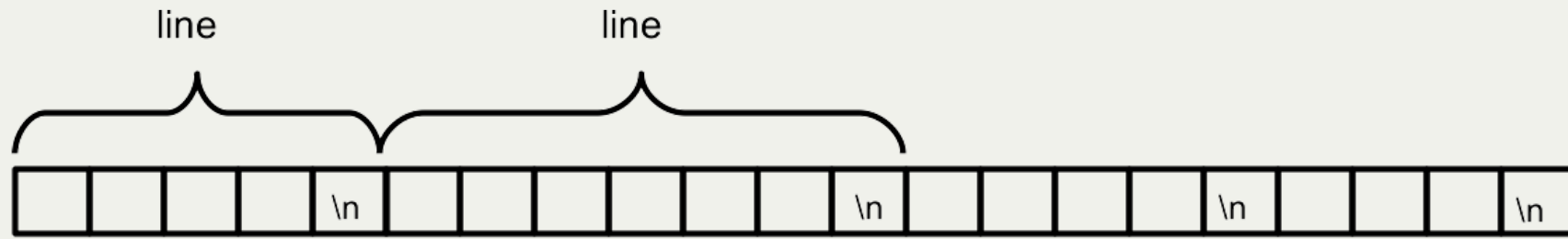
```
with open("somefile") as file_in:
```

# Reading one line at a time



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

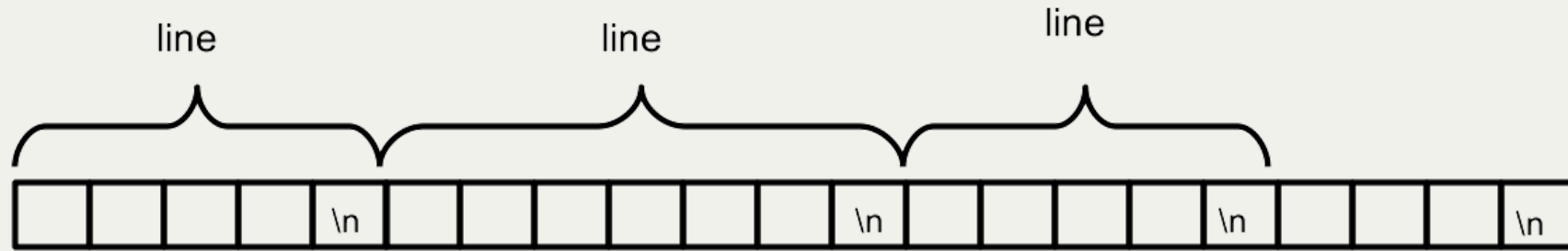
# Reading one line at a time



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

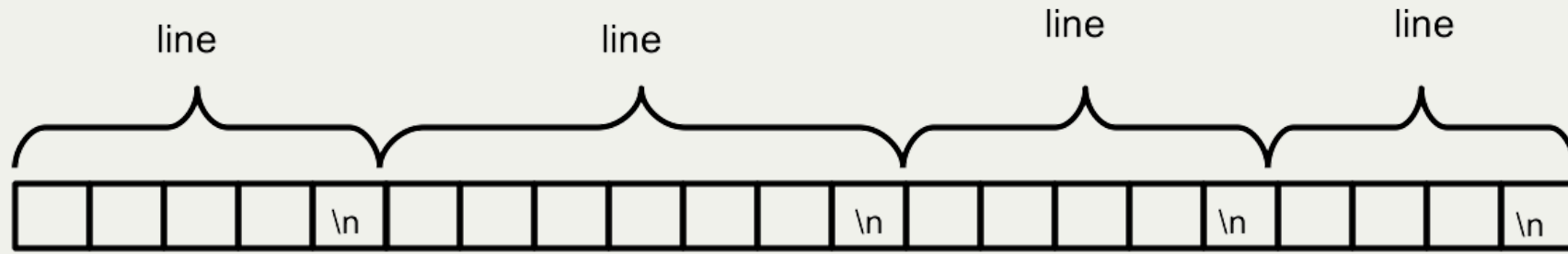


# Reading one line at a time



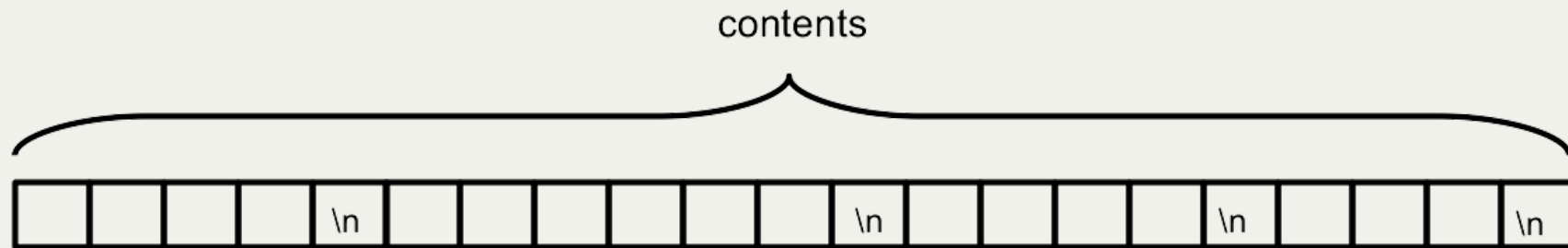
```
with open("somefile") as file_in:  
    for raw_line in file_in:  
        ...
```

# Reading one line at a time



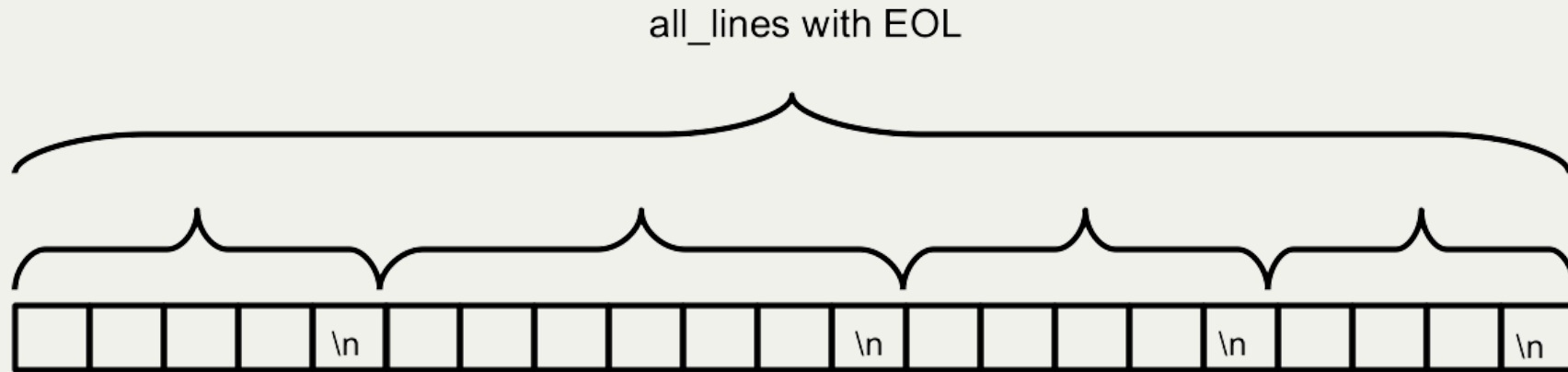
```
with open("somefile") as file_in:  
    for raw_line in file_in:  
        ...
```

# Reading entire file into string



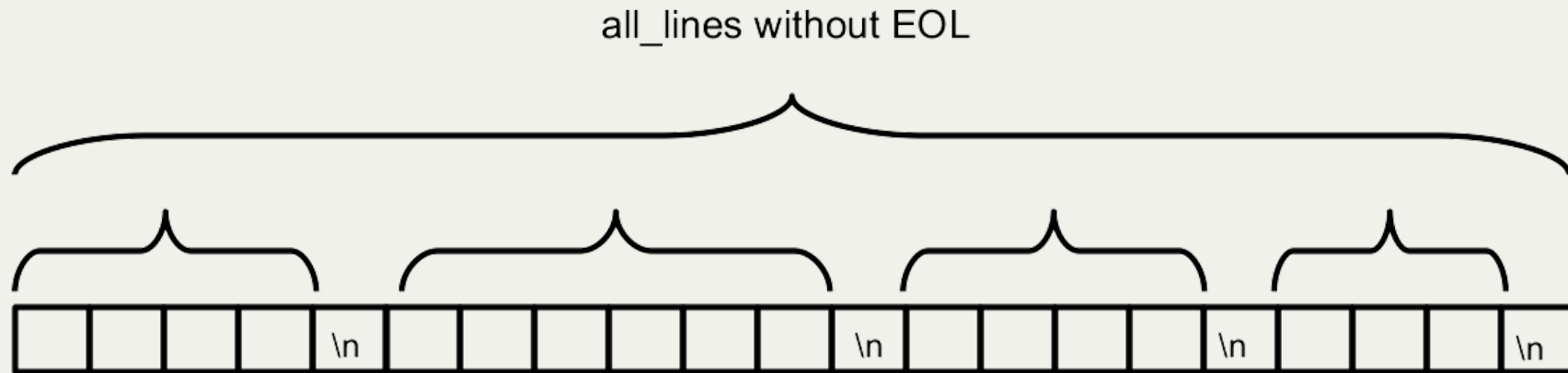
```
with open("somefile") as file_in:  
    contents = file_in.read()
```

# 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

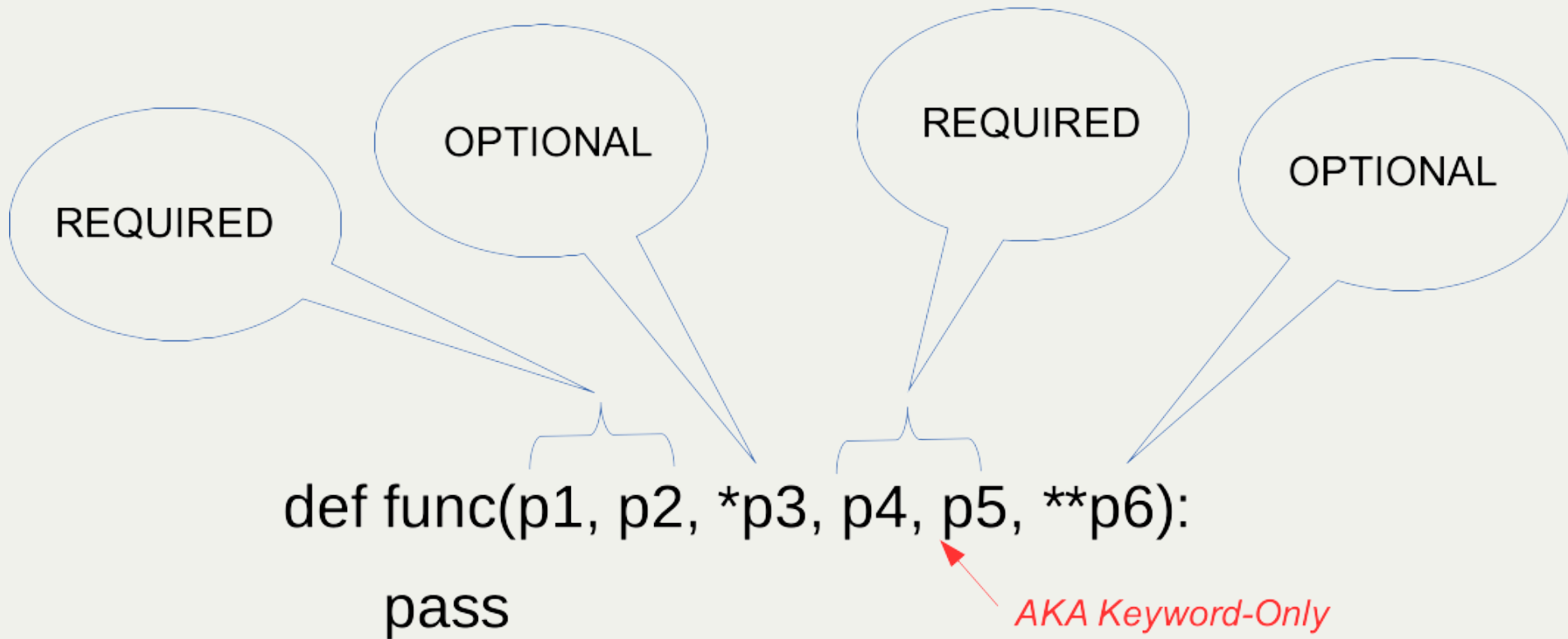
# Dictionary items

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

# Function parameters

POSITIONAL

NAMED

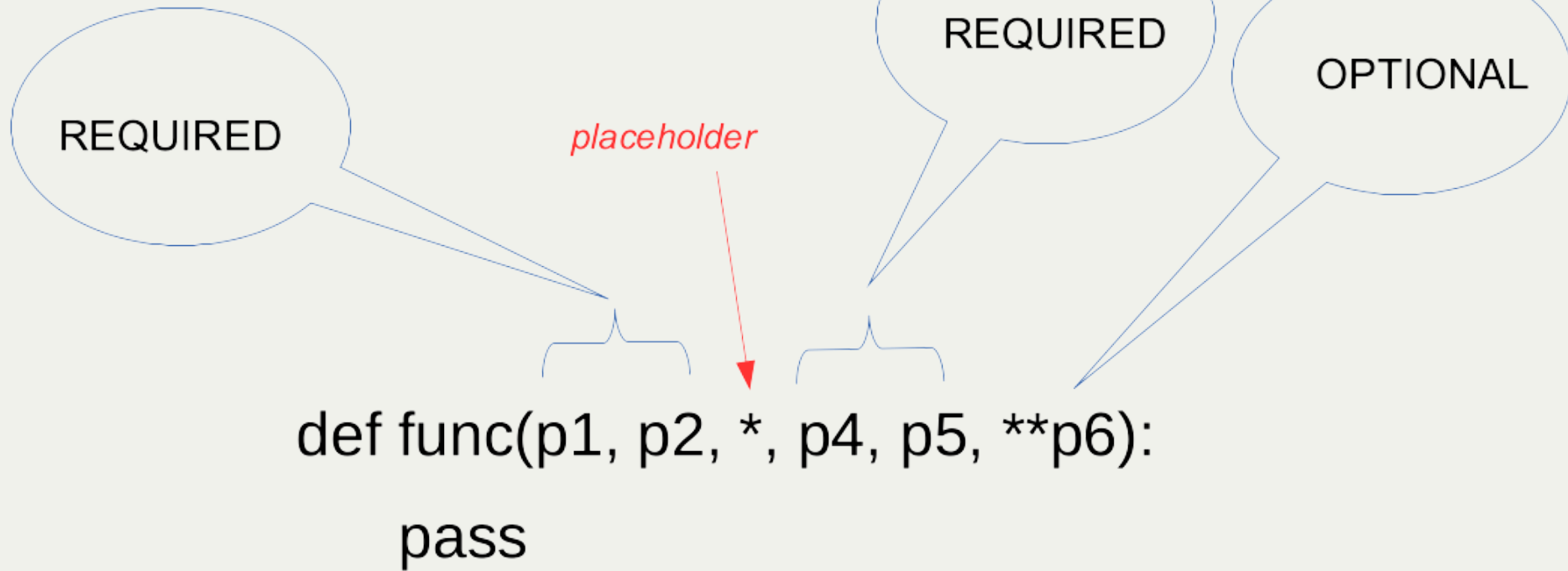




# Function parameters

POSITIONAL

NAMED



# Variable Scope

*builtin*

`print()`  
`len()`

*global*

`COUNT = 0`  
`LIMIT = 1`

*local*

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

# Variable scope

```
ALPHA = 10

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

spam(1234)
```

BUILTIN

GLOBAL

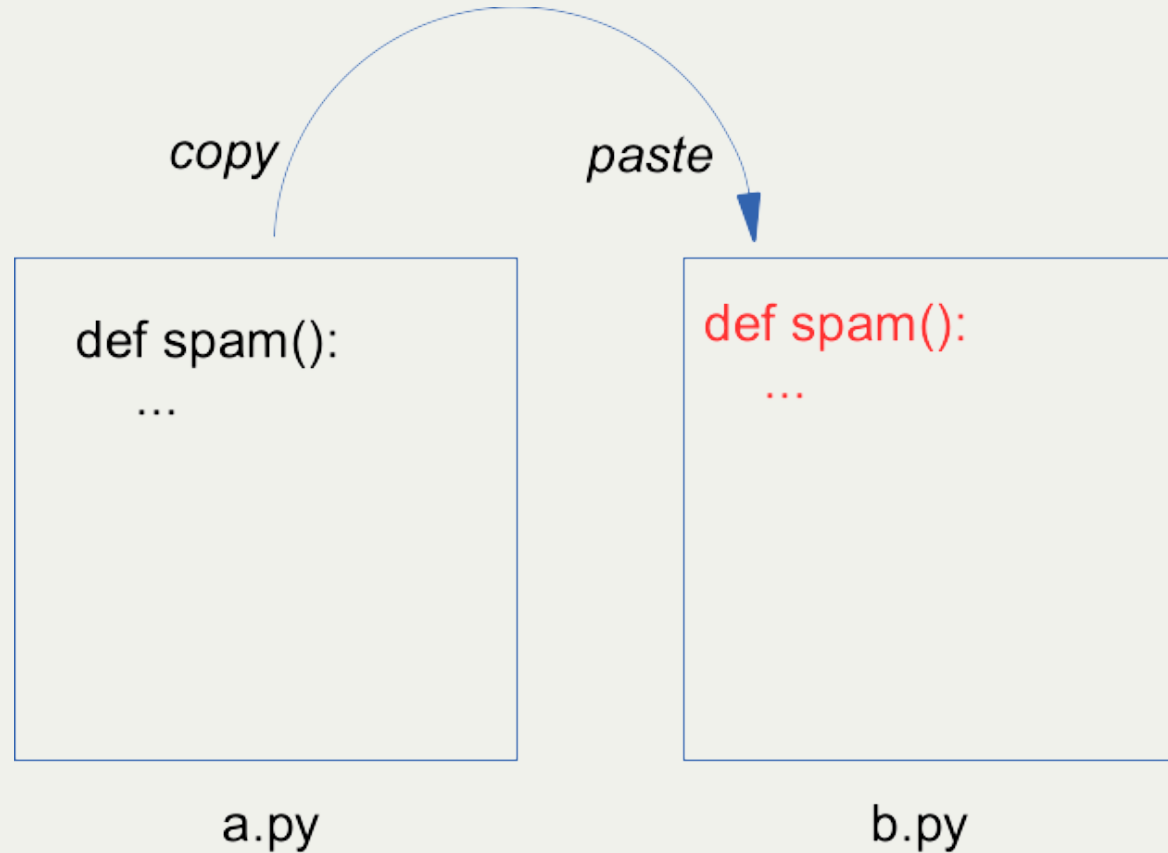
LOCAL

# Copy / pasting functions

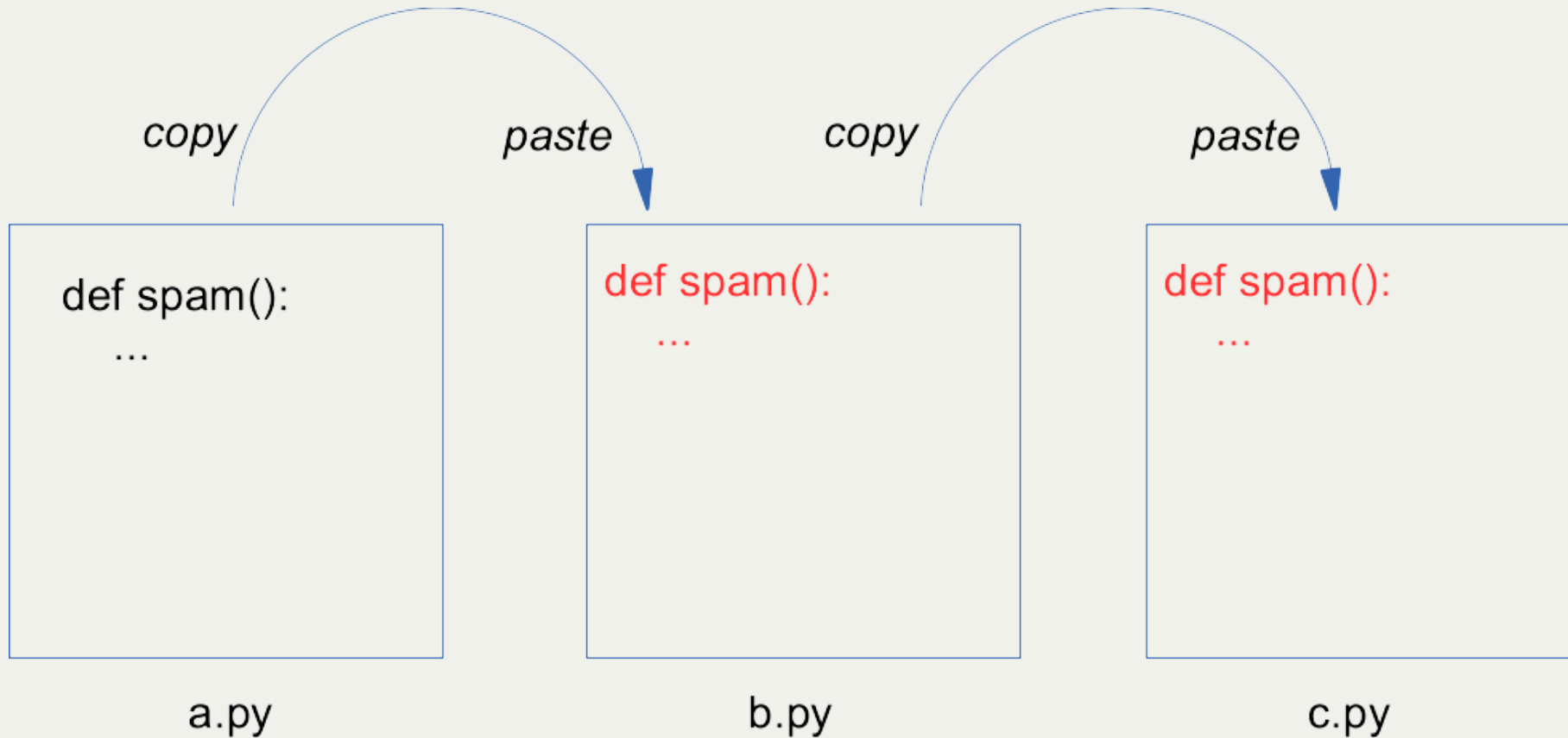
```
def spam():  
    ...
```

a.py

# Copy / pasting functions

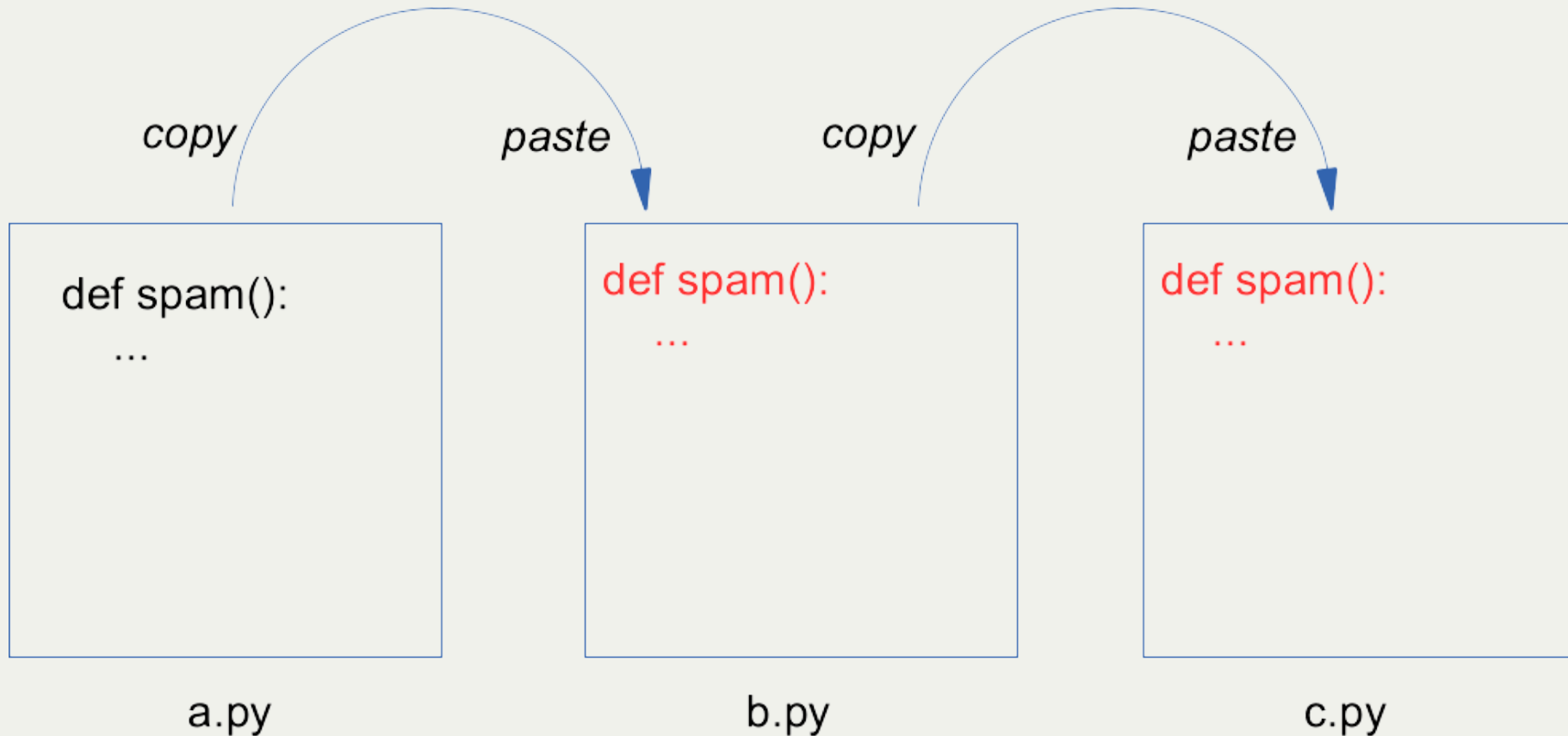


# Copy / pasting functions

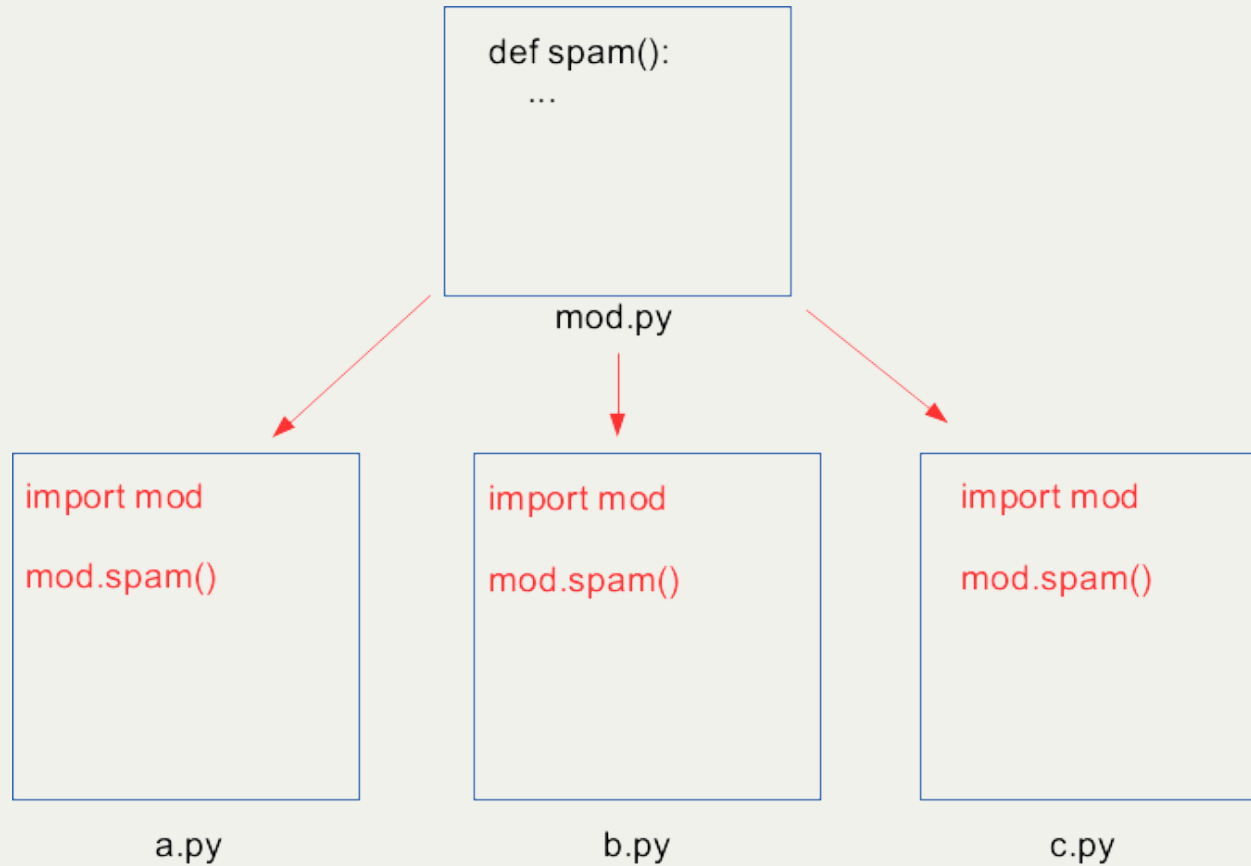


# Copy / pasting functions

**DON'T DO THIS!!**



# Using a module





# 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
```

# Advantages of Classes

## **Encapsulation**

Keep data and functions (methods) together

## **Keeping state**

Data persists between method calls

## **Readability**

Class attributes easier than lists/dicts

# Advantages of Classes

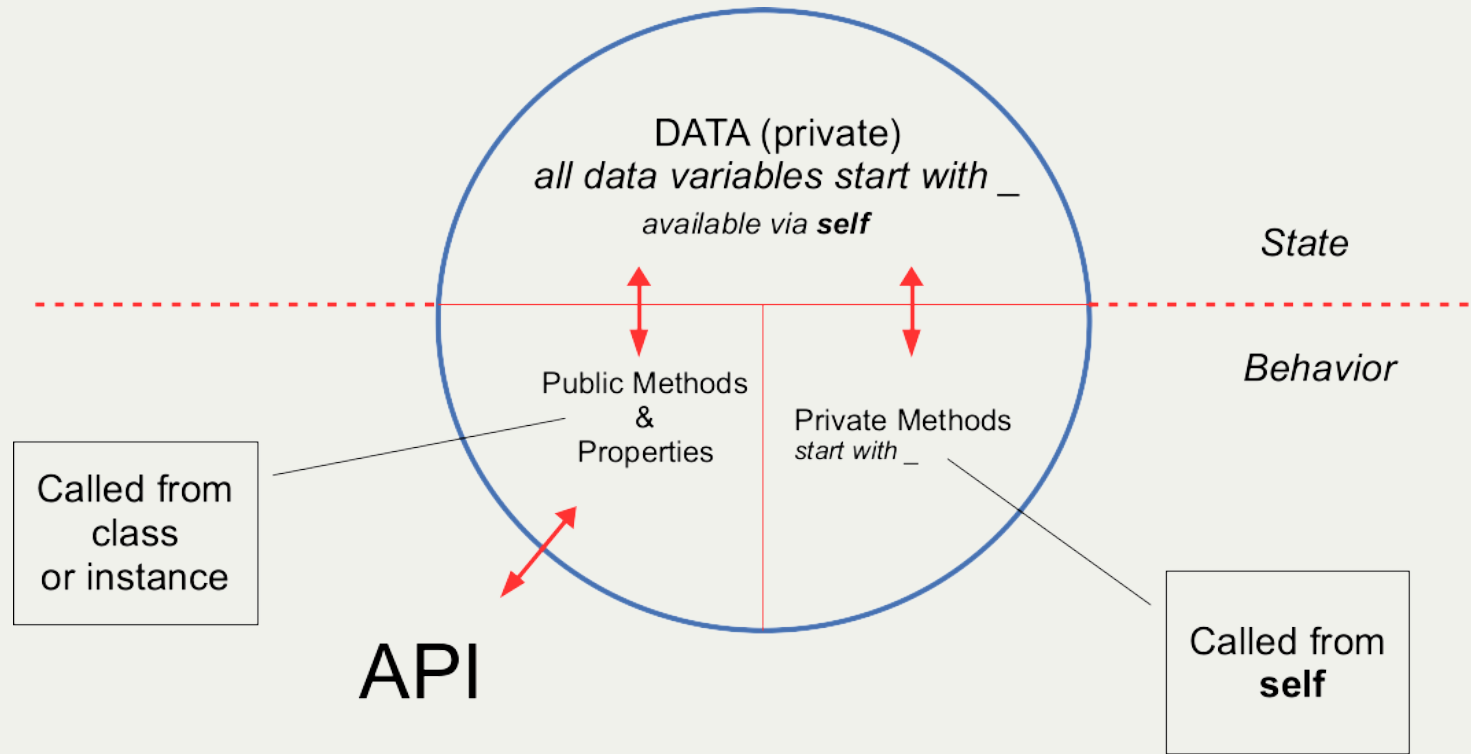
## **Convenience**

Import one class rather than multiple functions;  
methods autocomplete from objects

## **Inheritance**

Reuse and refactor code

# A Python Class



# Sorting

## Numbers

`n, n, n, ...`

## Strings

`"C1C2C3", "C1C2C3", "C1C2C3",`

## Nested iterables

`[obj1, obj2, obj3], [obj1, obj2, obj3],`

## Dictionary elements

`(key, value), (key, value), (key, value),`



# 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<sub>1</sub> | Branch<sub>2</sub>

Atom<sub>1</sub>Atom<sub>2</sub>Atom<sub>3</sub>(Atom<sub>4</sub>Atom<sub>5</sub>Atom<sub>6</sub>)Atom<sub>7</sub>

A a 1 ;

. \d \w \s  
[abc]  
[^abc]

Atom<sub>repeat</sub>

# 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)`**

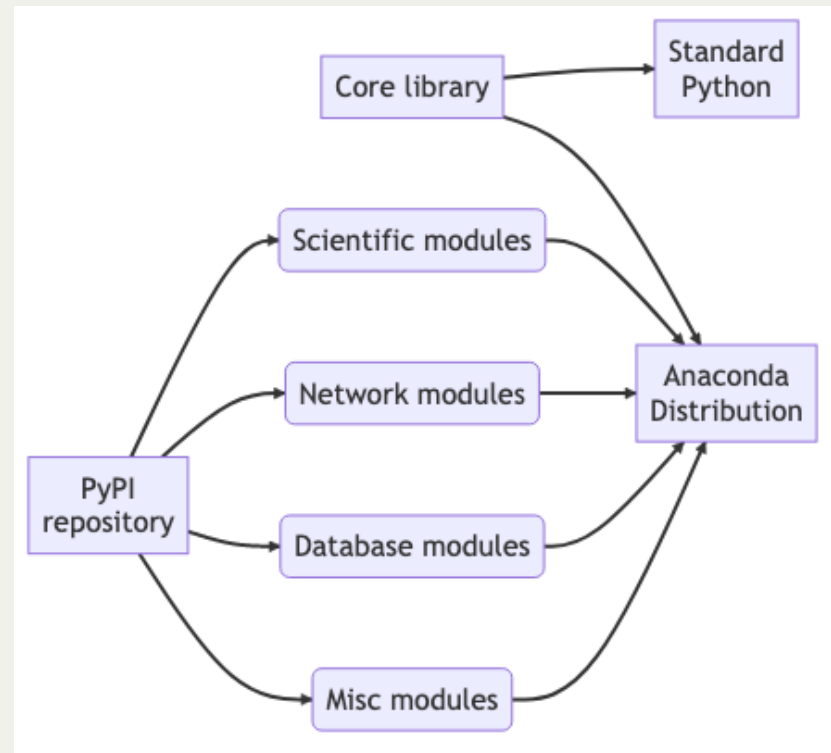
Split **text** using **re** as delimiter and return tokens as list.

# Guido van Rossum





# Available modules



# Advantages of Python

- Easy to learn
- Readable
- Modular
- Large Standard library
- Many third-party modules  
(science, data, web, admin, ...)

# Advantages of Python

- Multi-paradigm
  - Procedural
  - Object-oriented
  - Functional
- Fun!

# Disadvantages of Python

# What can Python do?

- Web apps
- Web services (REST, SOAP)
- Data mining / web scraping
- Data science
- End-user GUI apps
- System Administration (Windows, Mac, Linux)

# What can Python do?

- Scientific/Engineering analysis
- Data visualization
- Cloud apps