Coding Bootcamp Code in Python

PYTHON FUNDAMENTALS CONTINUED

Some more `str` methods: `strip`

- Getting rid of line endings:
  `rstrip('\r\n')`
  - method will strip all combinations of `\r` and `\n` from right end of string
  - Similar methods:
    - `lstrip`: strips from left end of string
    - `strip`: strips from both ends of string
  - no arguments, strips: space, `\t`, `\r`, `\n`, …

Note that strings are not modified, new string is created!

`str` method: `split`

- Splitting string: `split()` returns list of strings
  - no argument: split on (multiple) whitespace
  - otherwise, split on provided string
  - limit number of splits by providing extra argument
- E.g., read file, and print only end times

```
start 1: 2013-03-27 14:20:13
end 1: 2013-03-28 03:05:57
start 2: 2013-03-28 04:30:17
start 3: 2013-03-28 04:30:17
end 2: 2013-03-28 05:45:17
end 3: 2013-03-28 09:15:38
…
```

Split on `':'`, but note time format!!!

More `str` methods:
`startswith, endswith`

- `startswith(prefix),endswith(suffix)` return `True` if `str` starts with `prefix`/ends with `suffix` respectively, `False` otherwise

```
…
for line in sys.stdin:
    event_str = line.strip()
    if event_str.startswith('end'):
        event, time_str = event_str.split(':', 1)
        print(time_str)
```

Only single split, otherwise time is split as well

Even more `str` methods: `is<something>`

- Test `str` is uppercase/lowercase: `s.isupper()/s.islower()`
  - `'ABC'.isupper()  == True`
  - `'A19'.isupper()  == True`
  - `'Abc'.isupper()  == False`
  - `'19'.isupper()   == False`
- Test `str` has only whitespace: `s.isspace()`
- Test `str` has only digits: `s.isdigit()`
- Test `str` is alphabethic/alphanumeric: `s.isalpha()/s.isalnum()`

Searching & replacing in `str`

- Does `str` contain substring?
  `('ab' in 'ABabCD') == True`
- Find position of first occurrence of substring in `str`:
  `'ABabCD'.find('ab') == 2`
  - returns `-1` when not found
  - can search between given start and final position
- Replace all occurrences of substring by other substring
  `'3.14'.replace('.', ',') == '3,14'`
  - maximum number of replacements can be specified

More methods, but this will do

`str` operations

- Concatenating strings:
  `'abc' + 'def' == 'abcdef'`
  - Works for `list` as well
    `[0, 1] + [3, 4] == [0, 1, 3, 4]`
- Multiplying strings:
  `'x' * 4 == 'xxxx'`
  - Works for `list` as well
    `[0.0] * 4 == [0.0, 0.0, 0.0, 0.0]`
    - However, bear in mind that this may *not* always do what you think

Joining list elements

- Often, data contained in list data structure
  - Needs to be represented as delimited string
  - Example:

```
    [3.1745, 18.14, -6.49043]
→    3.1745,18.14,-6.49043
```

- Use list comprehension, `str` function and `str`'s `join(…)` method

```
>>> data = [3.1745, 18.14, -6.49043]
>>> print(','.join([str(number) for number in data]))
3.1745,18.14,-6.49043
```

type

`str` & `list` are sequences

- characters (elements for `list`) accessed by position, e.g., `s = 'abc'`:
  `s[0] == 'a',s[2] == 'c',`
  `s[-1] == 'c',s[-2] == 'b'`
- Substrings (slices for `list`), e.g.,
  `s = 'abcde'`:
  `s[0:3] == 'abc',s[2:4] == 'cd',`
  `s[1:] == 'bcde',s[:3] == 'abc'`
  `s[::-1] == 'edcba'`

`str` & `list` length revisited

- Function `len()` computes `str` length (number of elements for `list`)

  `len('') == 0`, `len('abc') == 3`

  `len([]) == 0`, `len([3, 5, 7]) == 3`
- Length & truth
  - Empty string is `False`, non-empty string `True`
  - Empty list is `False`, non-empty list is `True`

```
…
if len(line.strip()) > 0:
    …
```

≡

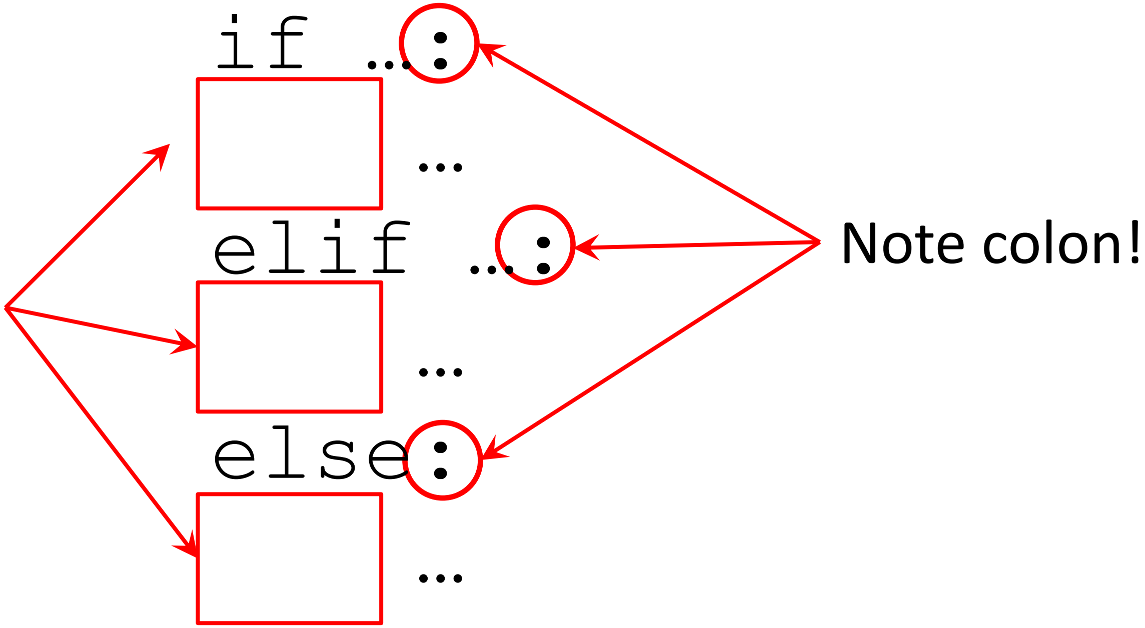
```
…
if line.strip():
    …
```

Type conversion

- Convert `str` `s` to floating point: `float(s)`
  - necessary for comparison:
    `float(data[2]) < 0.0`
- Convert `str` `s` to integer: `int(s)`
- Convert number `x` to `str`: `str(x)`
- Convert float `x` to integer: `int(x)`
  - takes integer part of float, e.g., `int(-3.8) == -3`
- Determining type of expression `e`: `type(e)`
  - e.g., `type(3 + 0.1) == float`

`if` statement

- General form:



Note indentation

Note colon!

- Nesting: structure through indentation
- Conditional expression:

```
…
n = 10 if r > 1.0 else 0
…
```

Conditionals

- Boolean values: `True`, `False`
- Boolean operators: `not`, `and`, `or`
- Comparison operators: `==`, `!=`, `<`, `<=`, `>`, `>=`
  - work on `str`, `float`, `int`,…
- List membership: `in`, e.g.,
  - `'a' in ['c', 'a', 'd'] == True`
  - `'e' not in ['c', 'a', 'd'] == True`

Which dimension numbers?

- Which dimension numbers occur in file?

```
import sys
def main():
    sys.stdin.readline()
    dim_nrs = set()
    for line in sys.stdin:
        dim_nrs.add(int(line.rstrip('\r\n').split()[1]))
    print(dim_nrs)
    return 0
```

Yuck, what's that?!?

```
dim_nr = int(line.rstrip('\r\n').split()[1])
```

⦀

```
line_str = line.rstrip('\r\n')
data     = line_str.split()
dim_str  = data[1]
dim_nr   = int(dim_str)
```

Python can be terse, but stick to what's comfortable for you!

However, use functions...

Reasonable compromise

- One additional variable simplifies code

```
import sys
def main():
    sys.stdin.readline()
    dim_nrs = set()
    for line in sys.stdin:
        data = line.rstrip('\r\n').split()
        dim_nrs.add(int(data[1]))
    print(dim_nrs)
    return 0
```

Sets

- `set` is Python data type, acts like set in math
  - empty set: `s = set()`
  - number of elements: `len(s)`
  - add element: `s.add('a')`
  - check membership: `'b' in s`
  - remove element: `s.remove('b')`, `s.discard('b')`
  - remove and return arbitrary element: `s.pop()`
  - iterating over elements:

    ```
    for element in s:
        …
    ```
- No `set` of `set`s, `set` of `list`s
- Set comprehensions:

  `{i for i in range(3)} ≡ {0, 1, 2}`

Set operations

```
s1 = {3, 5, 7}
s2 = {7, 11}
```

- Intersection: `s1 & s2`
  `s1.intersection(s2) == {7}`
- Union: `s1 | s2`
  `s1.union(s2) == {3, 5, 7, 11}`
- Difference: `s1 - s2`
  `s1.difference(s2) == {3, 5}`
- Symmetric difference: `s1 ^ s2`
  `s1.symmetric_difference(s2) == {3, 5, 11}`
- Is subset of? `s1 <= s2`
  `s1.issubset(s2) == False`
- Is disjoint from?
  `s1.isdisjoint(s2) == False`

To modify set, use:
`s1.<op>_update(s2)`
For union, use:
`s1.update(s2)`

`s1 |= s2`

More modularity

- Same code copied and pasted, modified

```
…
for line in sys.stdin:
    data = line.rstrip('\r\n').split()
    dim_nr = int(data[1])
    …
```

- Make it generic

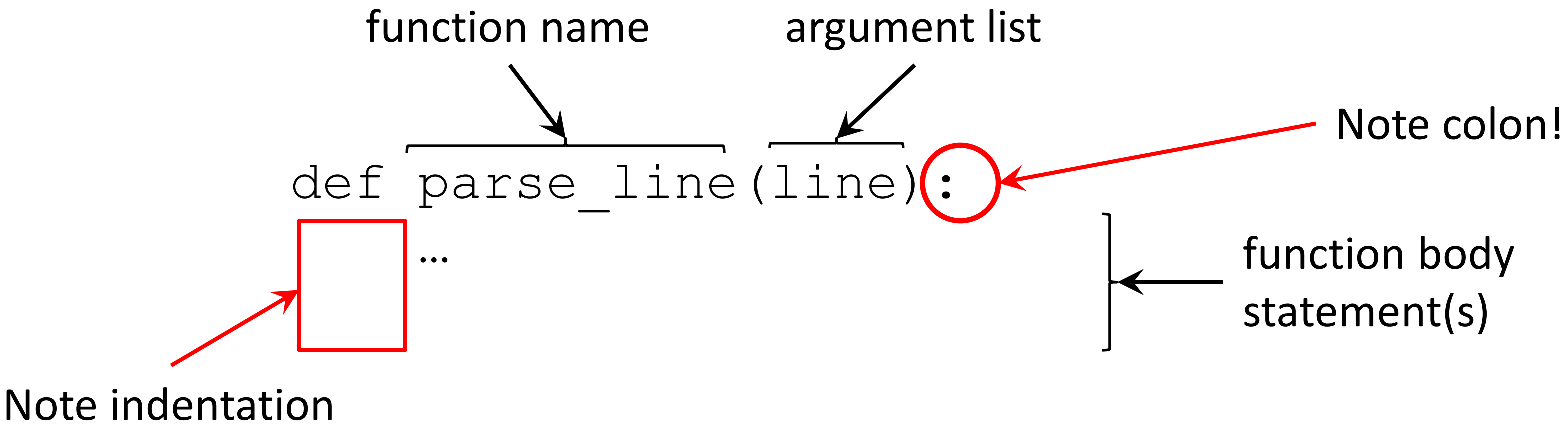
```
def parse_line(line):
    data = line.rstrip('\r\n').split()
    return (int(data[0]), int(data[1]), float(data[2]))
…
for line in sys.stdin:
    case_nr, dim_nr, temp = parse_line(line)
    …
```

Functions

- Call by reference
  - however, remember that `str`, `int`, `float` et al. are immutable
- Arguments can have default values
- Arguments can be positional, or by keyword
- Higher order
  - functions can have functions as arguments
  - function can return functions (closures)

Anatomy of function definition

- Function definition



function name

argument list

```
def parse_line(line):
    …
```

Note colon!

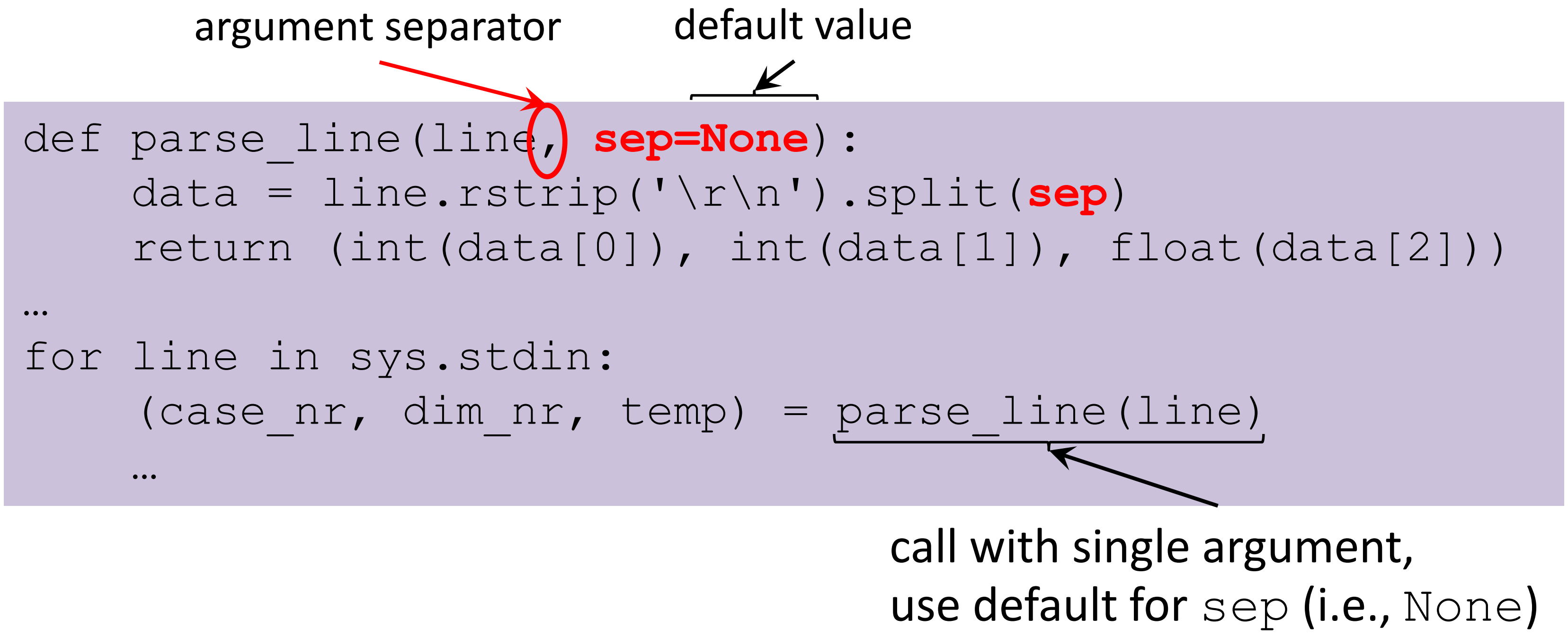
function body statement(s)

Note indentation

- `return` statement to… return results, if any, and return control to caller

Adding flexibility

- Optional column separator



argument separator

default value

```
def parse_line(line, sep=None):
    data = line.rstrip('\r\n').split(sep)
    return (int(data[0]), int(data[1]), float(data[2]))
…
for line in sys.stdin:
    (case_nr, dim_nr, temp) = parse_line(line)
    …
```

call with single argument,
use default for sep (i.e., None)

```
    …
    (case_nr, dim_nr, temp) = parse_line(line, sep=';')
    …
```

Default value pitfall

```
def filter_pos(new_values, values=[]):
    for new_value in new_values:
        if new_value > 0:
            values.append(new_value)
    return values

if __name__ == '__main__':
    value_list = [
        [1, -3, 5],
        [13, 33, -15],
    ]
    for values in value_list:
        print(f'filtering {values}')
        filtered_values = filter_pos(values)
        print(f'filtered: {filtered_values}')
```

default values are
created on import,
reused for calls:
mutable types == surprise!

```
filtering [1, -3, 5]
filtered: [1, 5]
filtering [13, 33, -15]
filtered: [1, 5, 13, 33]
```

```
def filter(new_values, values=None):
    if values is None:
        values = []
    …
```

Tuples (YADS ☺)

- `tuple` is (kind of) fixed length list, *immutable*
- `tuple` with two elements: `t = ('a', 'b')`
  - first element: `t[0] == 'a'`, second element `t[1] == 'b'`

```
def parse_line(line, sep=None):
    data = line.rstrip('\r\n').split(sep)
    return (int(data[0]), int(data[1]), float(data[2]))
…

for line in sys.stdin:
    case_nr, dim_nr, temp = parse_line(line)
    …
```

`tuple` of `int`, `int`, `float`

3-`tuple` unpacked into 3 variables

1-`tuple`: `('a', )`

Returning to dimension numbers…

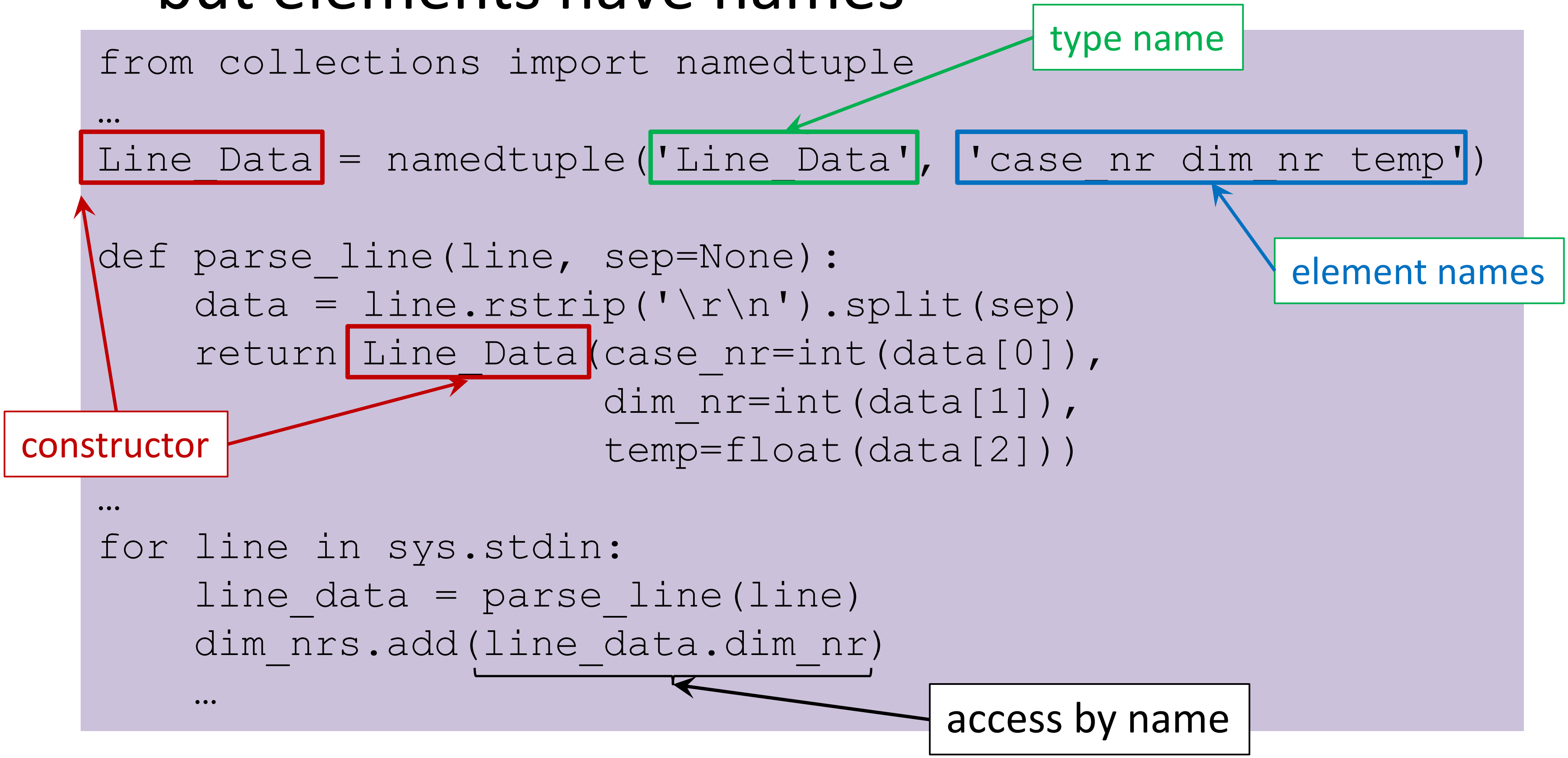
- Which dimension numbers occur in file?

```
…
def main():
    _ = sys.stdin.readline()
    dim_nrs = set()
    for line in sys.stdin:
        _, dim_nr, _ = parse_line(line)
        dim_nrs.add(dim_nr)
    for dim_nr in dim_nrs:
        print(dim_nr)
```

_ is wildcard in tuple unpacking:
tuple elements at those positions are ignored

Named tuples, Python 2.6+

- `collections.namedtuple` *is* `tuple`, but elements have names



```
from collections import namedtuple
…
Line_Data = namedtuple('Line_Data', 'case_nr dim_nr temp')

def parse_line(line, sep=None):
    data = line.rstrip('\r\n').split(sep)
    return Line_Data(case_nr=int(data[0]),
                     dim_nr=int(data[1]),
                     temp=float(data[2]))
…
for line in sys.stdin:
    line_data = parse_line(line)
    dim_nrs.add(line_data.dim_nr)
    …
```

type name

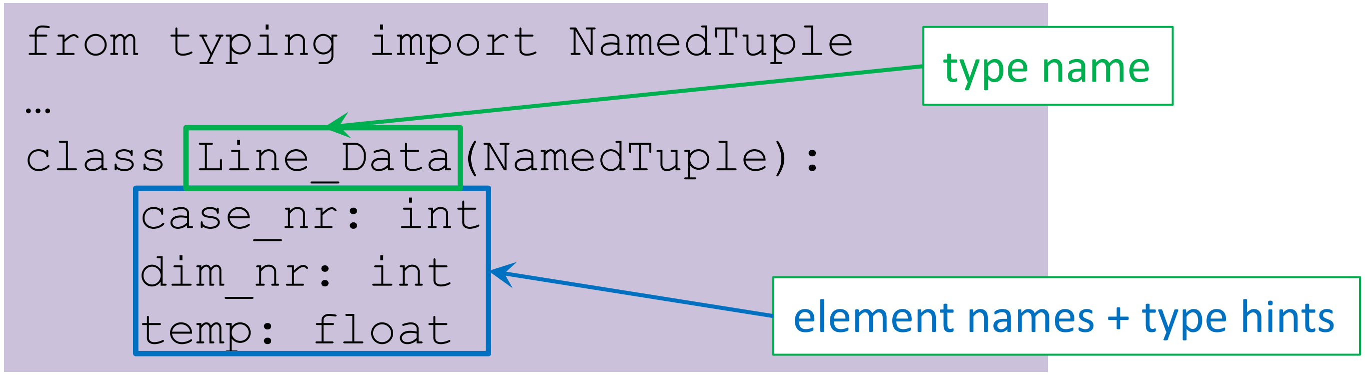
element names

constructor

access by name

Named tuples, Python 3.5+

- `typing.NamedTuple` ***acts as*** `tuple`, but
  - elements have names
  - elements have type hints
  - can have methods
  - can serve as base class



```
from typing import NamedTuple
…
class Line_Data(NamedTuple):
    case_nr: int
    dim_nr: int
    temp: float
```

type name

element names + type hints

Using named tuples

```
…
def parse_line(line, sep=None):
    data = line.rstrip('\r\n').split(sep)
    return Line_Data(case_nr=int(data[0]),
                     dim_nr=int(data[1]),
                     temp=float(data[2]))
…
for line in sys.stdin:
    line_data = parse_line(line)
    dim_nrs.add(line_data.dim_nr)
    …
```

element values can be specified by name in any order

access by name

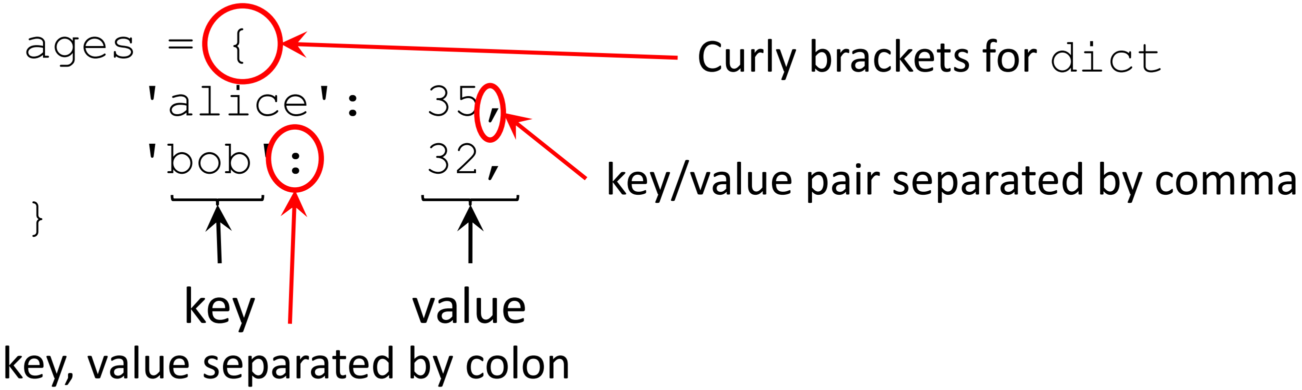
Counting dimension numbers

- How many times does a dimension number occur in file?
  - maximum & minimum not known a-priori!

```
…
import sys
def main():
    _ = sys.stdin.readline()
    counter = dict()
    for line in sys.stdin:
        line_data = parse_line(line)
        if line_data.dim_nr not in counter:
            counter[line_data.dim_nr] = 0
        counter[line_data.dim_nr] += 1
    for dim_nr, count in counter.items():
        print('{0}: {1}'.format(dim_nr, count))
```

Dictionaries

- Data structure that maps a key onto a value
  - e.g., map a name to an age



```
ages = {
      'alice':  35,
      'bob':    32,
}
```

Curly brackets for `dict`

key/value pair separated by comma

key

value

key, value separated by colon

  - Keys can have any (hashable) type (mixed too)
  - Values can have any type (mixed too)
  - Dictionary comprehensions:
    `{k: k**2 for k in range(3)}` ≡ `{0: 0, 1: 1, 2:4}`

Using dictionaries

```
ages = {
    'alice':  35,
    'bob':    32,
}
```

- Empty dictionary: `{}` or `dict()`
- Number of key/value pairs: `len(ages)`
- Storing values

  ```
  ages['caro'] = 45
  ```
- Retrieving values

  ```
  35 == ages['alice']
  ```
- Removing key/value, and return value

  ```
  35 == ages.pop('alice')
  ```
- Does `ages` have an age for `'dave'`?

  ```
  ages.has_key('dave') ≡ 'dave' in ages
  ```

Iterating over dictionaries

- Iterate over keys:

```
for name in ages.keys():
    …
for name in ages:
    …
```

- Iterate over values:

```
for age in ages.values():
    …
```

- Iterate over key/value pairs:

```
for name, age in ages.items():
    …
```

Note:
creates views

Python 3.6+ *implementation*: keys in insertion order!

Counting again…

- Using `collections.Counter` instead of `dict`: simpler, less error prone

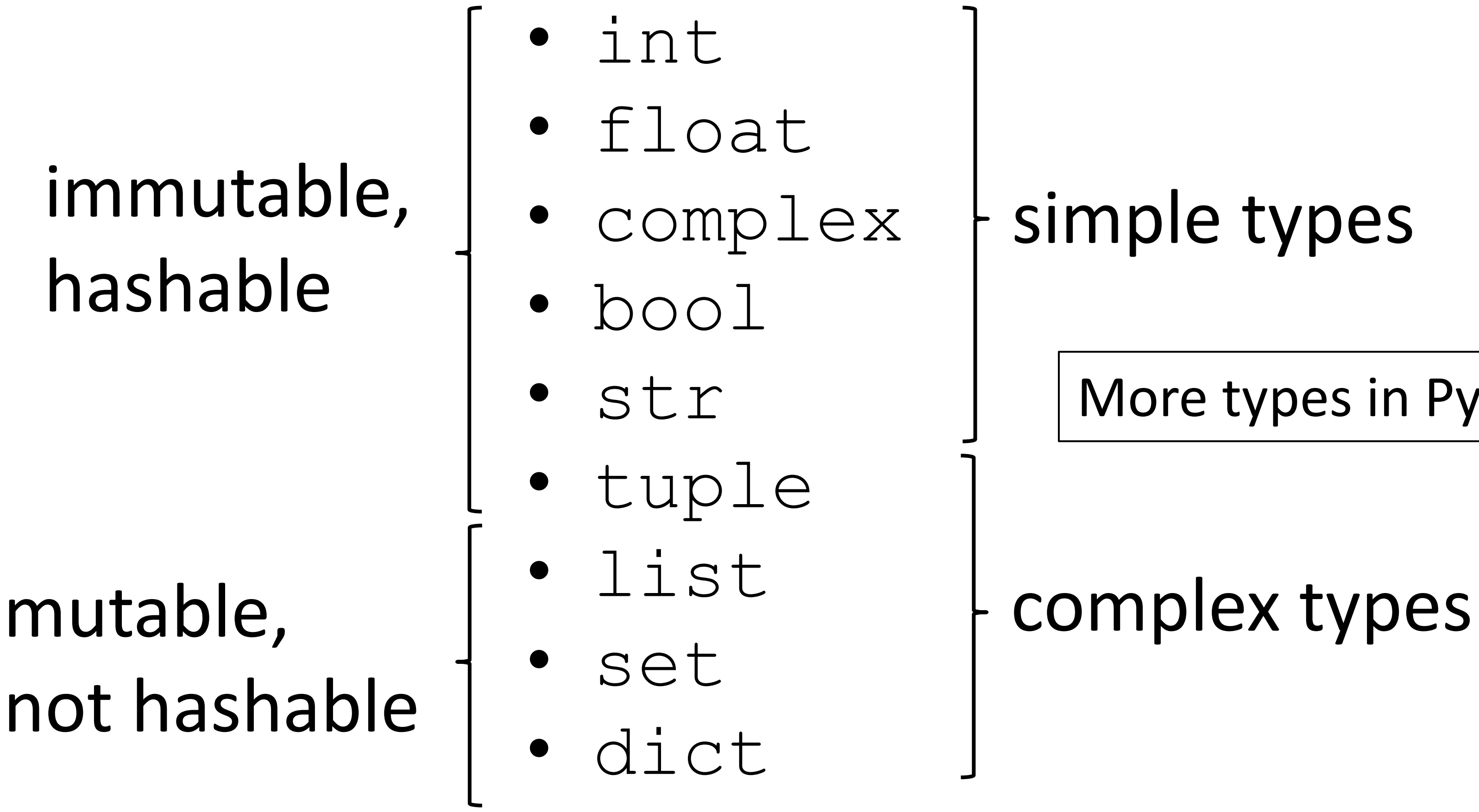
```
…
import collections
import sys
def main():
    _ = sys.stdin.readline()
    counter = collections.Counter()
    for line in sys.stdin:
        line_data = parse_line(line)
        counter[line_data.dim_nr] += 1
    for dim_nr, count in counter.items():
        print('{0}: {1}'.format(dim_nr, count))
```

bonus method: `most_common()`

More special data types

- `collections.namedtuple`: tuples with named elements
- `collections.Counter`: count elements
- `collections.OrderedDict`: remembers insertion order
- `collections.deque`: (bounded) double-ended queue
- `collections.defaultdict`: dictionary with computed default values
- `array.array`: faster than lists, however, use numpy

Summary: data types



immutable, hashable

- `int`
- `float`
- `complex`
- `bool`
- `str`
- `tuple`

mutable, not hashable

- `list`
- `set`
- `dict`

simple types

complex types

More types in Python

Picking the right data type is crucial to produce good code

Summary: control structures

- Conditional statement:

```
if …:
    …
elif …:
    …
else:
    …
```

- Iteration statements:
  - for-loop:

    ```
    for … in …:
        …
    ```

  - while-loop:

    ```
    while …:
        …
    ```

Summary: mathematics

- Usual operators: `+, -, *, /, %`
  - for `int`, division is floating point division, i.e., `3/5 == 0.6`
- Raise to power: `**`
  - e.g., `2**4 == 16`
- Floor division: `//`
  - e.g., `7.3//5.7 == 1.0`, but `6//4 == 1`
- Mathematical functions in module `math`
  - First import module (usually at top of file):
    ```
    import math
    ```
    Use functions, e.g., `math.sqrt(3.0)`
  - Or import specific function(s):
    ```
    from math import sqrt
    ```
    Use function(s), e.g., `sqrt(3.0)`
- For complex numbers, functions in `cmath`

changed from 2.x to 3.x!

Code Pack 04

A. Python fundamentals:

~~1. Primitive Datatypes and Operators~~

~~2. Variables and Collections~~

3. Control Flow and Iterables

4. Functions

B. Port Scanning

Coding Bootcamp Code in Python

CODE ORGANIZATION

Python modules & packages

- Code organization
  - Functions common to multiple scripts can be put in separate file = module
  - Modules can be organized hierarchically in directory structure = packages

Don't forget `__init__.py` in package directories!

- Python standard library is organized in packages

Example module & use

- Module file:

data_parsing.py

```
from collections import namedtuple
Line_Data = namedtuple('Line_Data', 'case_nr dim_nr temp')

def parse_line(line, sep=None):
    data = line.rstrip('\r\n').split(sep)
    return Line_Data(case_nr=int(data[0]),dim_nr=int(data[1]),
                     temp=float(data[2]))
```

- Using the module in script:

counting.py

```
import data_parsing
def main():
    …
    for line in sys.stdin:
        line_data = data_parsing.parse_line(line)
        …
```

Importing functions directly

- Importing function `parse_line` from module `data_parsing` in script `counting.py`:

counting.py

```
…
from data_parsing import parse_line

def main():
    …
    for line in sys.stdin:
        data = parse_line(line)
        …
```

More concise, but name clashes can occur!
E.g., `math.sqrt` versus `cmath.sqrt`

```
from math import sqrt
from cmath import sqrt as csqrt
```

Double duty

data_parsing.py

```
from collections import namedtuple
Line_Data = namedtuple('Line_Data',
                       ['case_nr', 'dim_nr', 'temp'])

def parse_line(line, sep=None):
    data = line.rstrip('\r\n').split(sep)
    return Line_Data(case_nr=int(data[0]),
                     dim_nr=int(data[1]),
                     temp=float(data[2]))

if __name__ == '__main__':
    …
    for line in sys.stdin:
        line_data = parse_line(line)
        …
```

Only executed when run as script

Package layout & use example

- weave.py
- vsc
    - __init__.py
    - util.py
    - parameter_weaver
        - __init__.py
        - artifact.py
        - base_formatter.py
        - c
            - __init__.py
            - formatter.py
            - …
        - fortran
            - __init__.py
            - formatter.py
            - …
        - …

```
…
from vsc.parameter_weaver.c.formatter import Formatter
…
```

```
…
from vsc.parameter_weaver.base_formatter import BaseFormatter
…
```

Code Pack 05

A. Enter VS Code

B. Python fundamentals:

~~1. Primitive Datatypes and Operators~~

~~2. Variables and Collections~~

~~3. Control Flow and Iterables~~

~~4. Functions~~

5. Modules