Brief Course in Python

Jian-hua Yeh

jhyeh@mail.au.edu.tw

智慧計算實驗室

SAIC Smart Computing

Outline

- The Basics
- The Not-So-Basics
- Object-oriented Programming

Getting Python

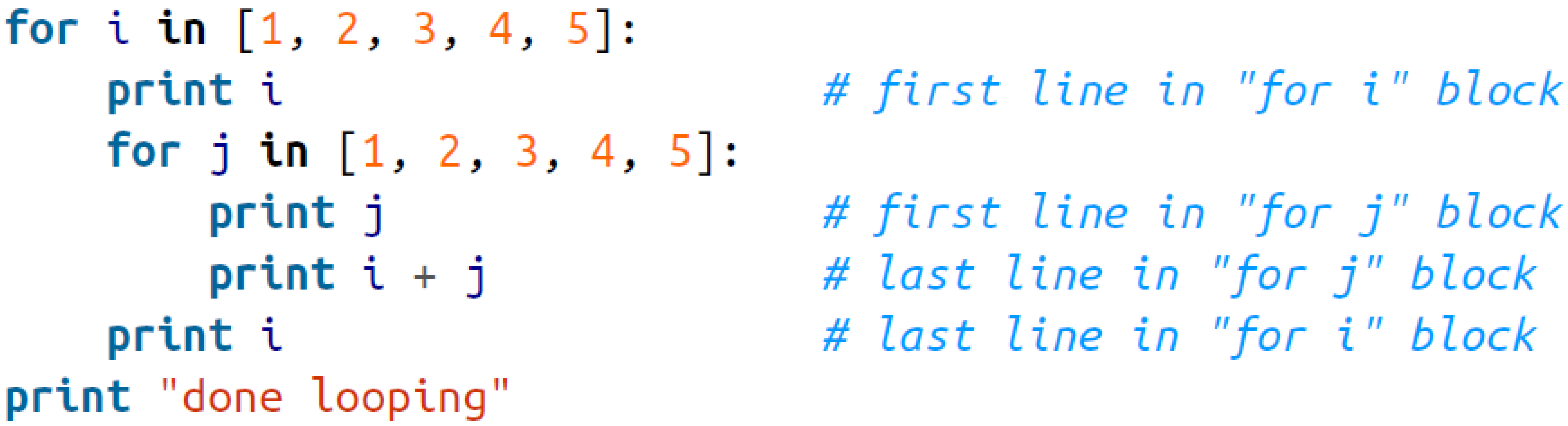
- Method 1: download Python from python.org
- Method 2: installing the Anaconda distribution (already includes most of the data science libraries)

Design Principle

There should be one - and preferably only one - obvious way to do it

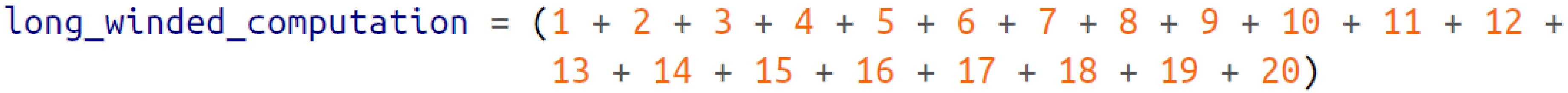
Whitespace Formatting

- Whitespace is ignored inside parentheses and brackets

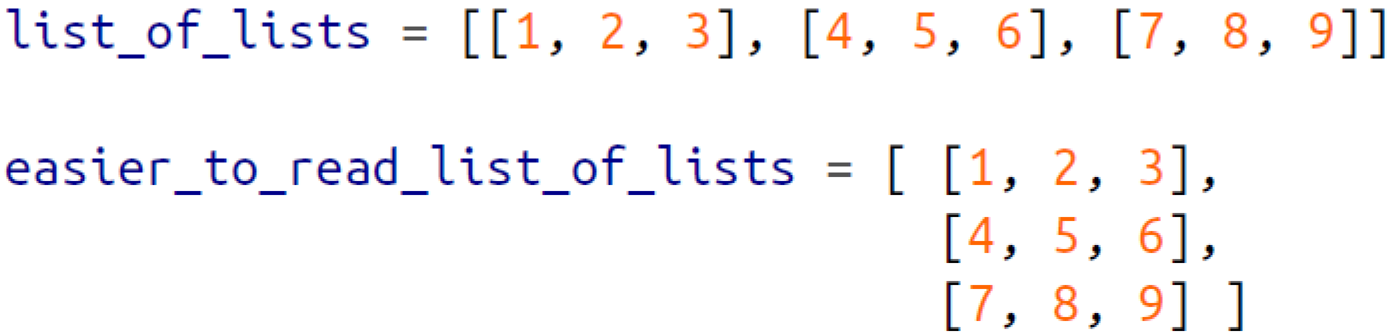


```
for i in [1, 2, 3, 4, 5]:
    print i                     # first line in "for i" block
    for j in [1, 2, 3, 4, 5]:
        print j                 # first line in "for j" block
        print i + j             # last line in "for j" block
    print i                     # last line in "for i" block
print "done looping"
```

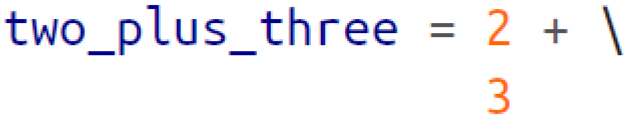
Whitespace Formatting



```
long_winded_computation = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 +
                           13 + 14 + 15 + 16 + 17 + 18 + 19 + 20)
```



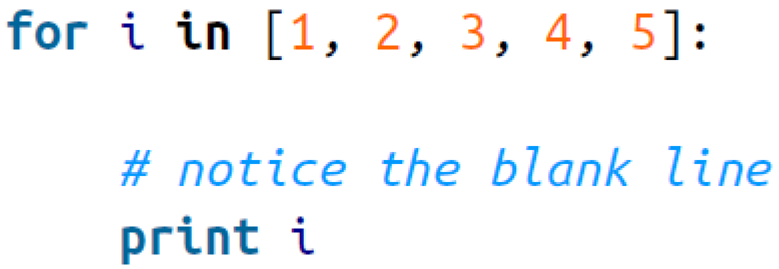
```
list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

easier_to_read_list_of_lists = [ [1, 2, 3],
                                 [4, 5, 6],
                                 [7, 8, 9] ]
```



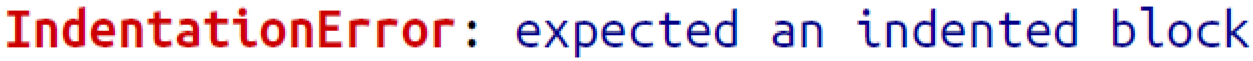
```
two_plus_three = 2 + \
                 3
```

Whitespace Formatting

- Blank line signals the end of the for loop’s block



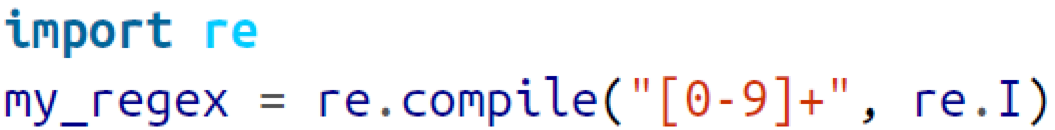
```
for i in [1, 2, 3, 4, 5]:

    # notice the blank line
    print i
```



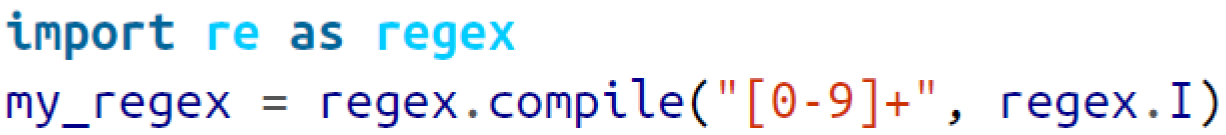
```
IndentationError: expected an indented block
```

Modules

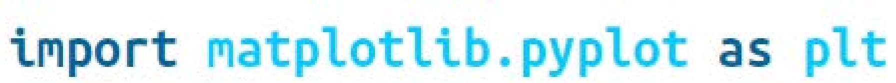
- Certain features of Python are not loaded by default



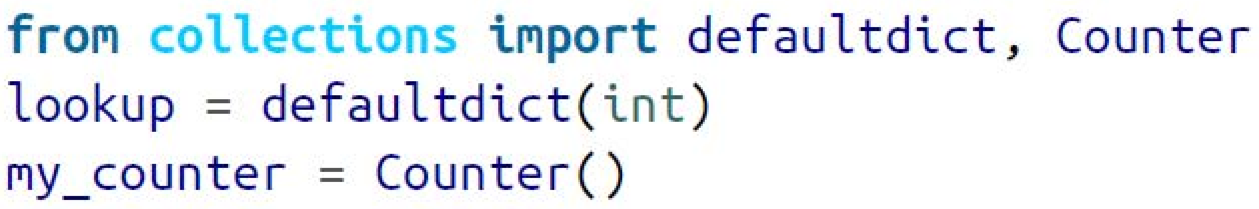
```python
import re
my_regex = re.compile("[0-9]+", re.I)
```



```python
import re as regex
my_regex = regex.compile("[0-9]+", regex.I)
```



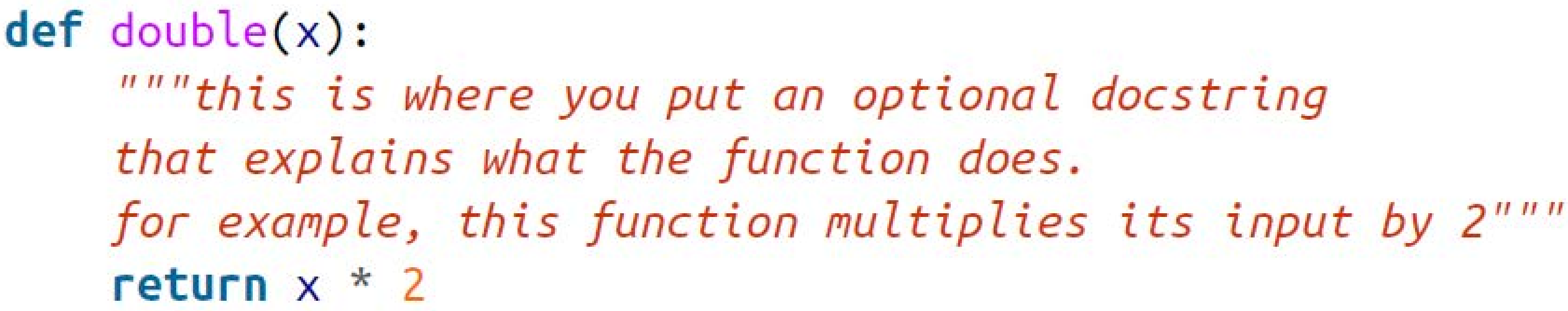
```python
import matplotlib.pyplot as plt
```



```python
from collections import defaultdict, Counter
lookup = defaultdict(int)
my_counter = Counter()
```

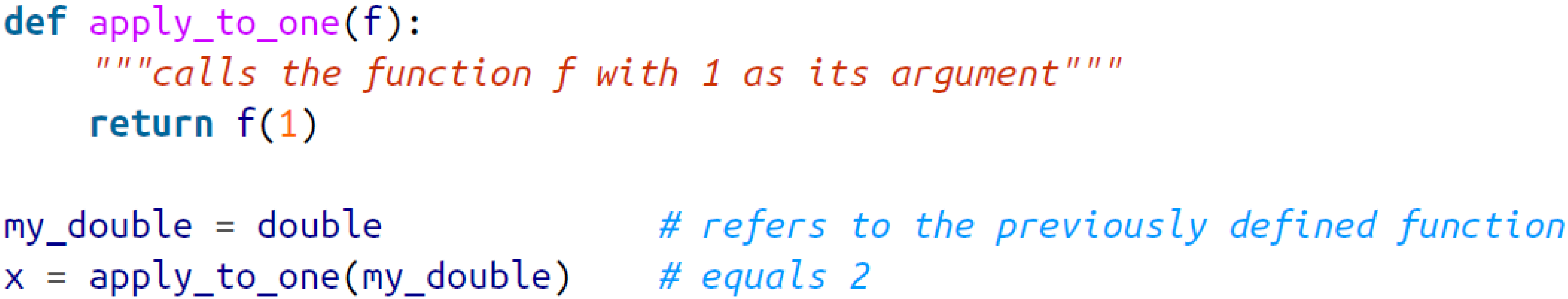
Functions

- A function is a rule for taking zero or more inputs and returning a corresponding output



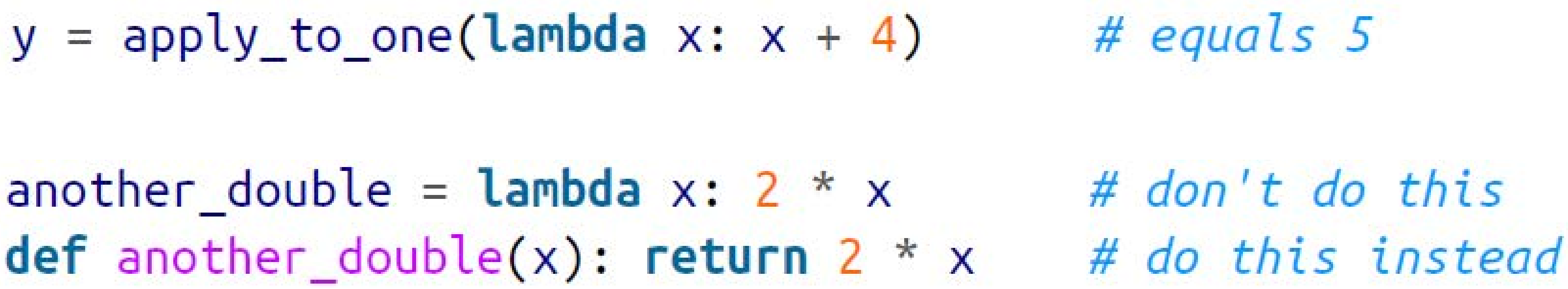
```python
def double(x):
    """this is where you put an optional docstring
    that explains what the function does.
    for example, this function multiplies its input by 2"""
    return x * 2
```

Functions



```python
def apply_to_one(f):
    """calls the function f with 1 as its argument"""
    return f(1)

my_double = double             # refers to the previously defined function
x = apply_to_one(my_double)    # equals 2
```

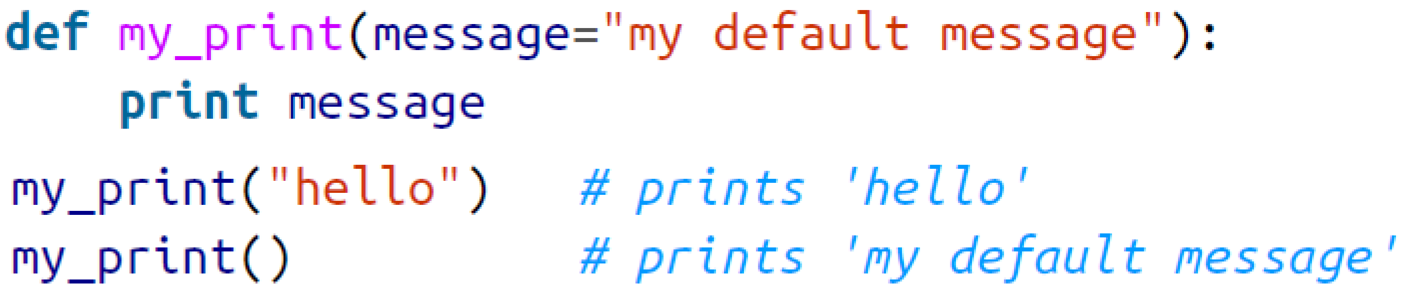
- Lambda/Anonymous function



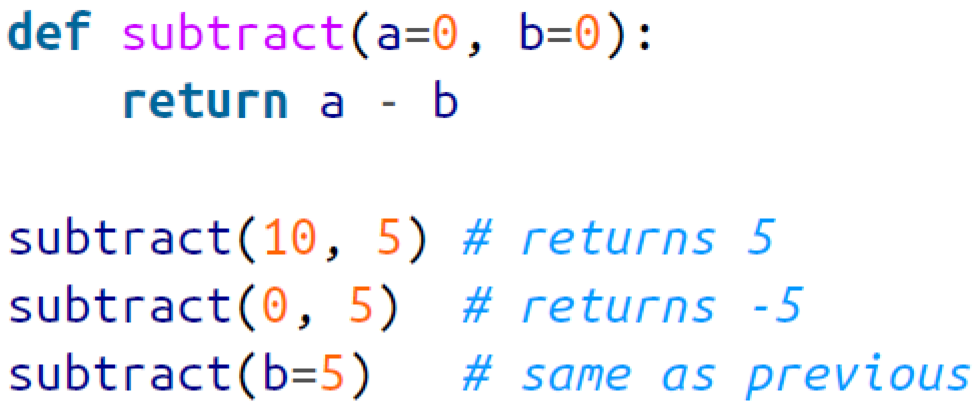
```python
y = apply_to_one(lambda x: x + 4)      # equals 5

another_double = lambda x: 2 * x       # don't do this
def another_double(x): return 2 * x    # do this instead
```

Functions

- Default arguments



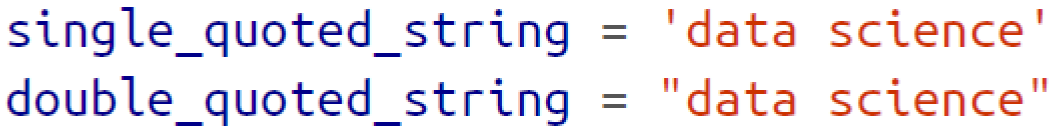
```
def my_print(message="my default message"):
    print message

my_print("hello")    # prints 'hello'
my_print()           # prints 'my default message'
```



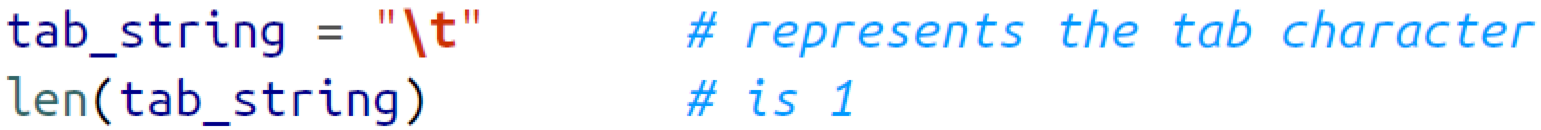
```
def subtract(a=0, b=0):
    return a - b

subtract(10, 5) # returns 5
subtract(0, 5)  # returns -5
subtract(b=5)   # same as previous
```

Strings

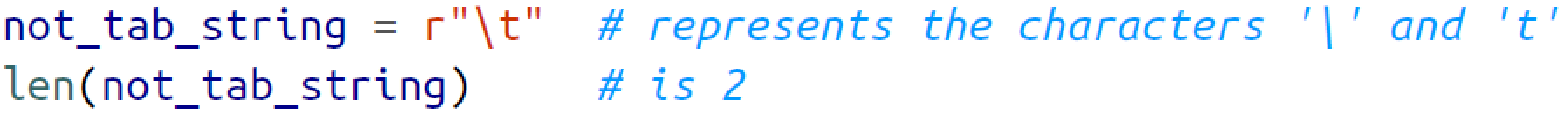
- Strings can be delimited by single or double quotation marks (but the quotes have to match)



```
single_quoted_string = 'data science'
double_quoted_string = "data science"
```



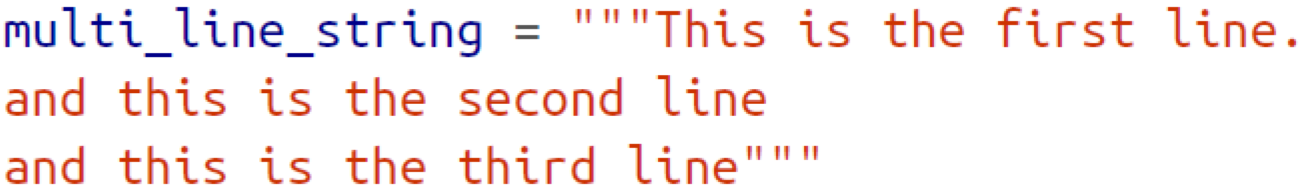
```
tab_string = "\t"       # represents the tab character
len(tab_string)         # is 1
```



```
not_tab_string = r"\t"  # represents the characters '\' and 't'
len(not_tab_string)     # is 2
```

Strings

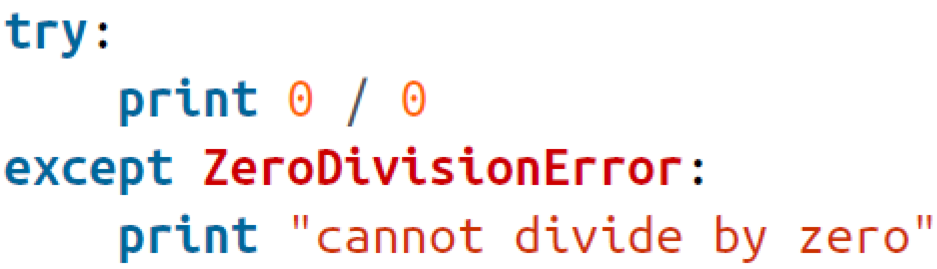
- Create multiline strings using triple-[double-]-quotes



```
multi_line_string = """This is the first line.
and this is the second line
and this is the third line"""
```

Exceptions

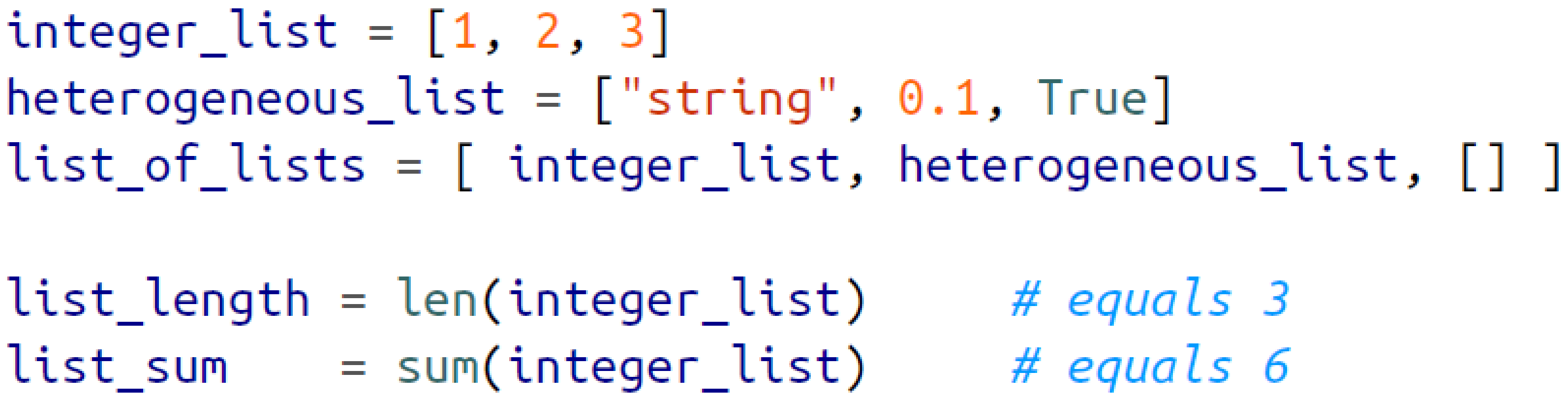
- When something goes wrong, Python raises an *exception*. Unhandled, these will cause your program to crash



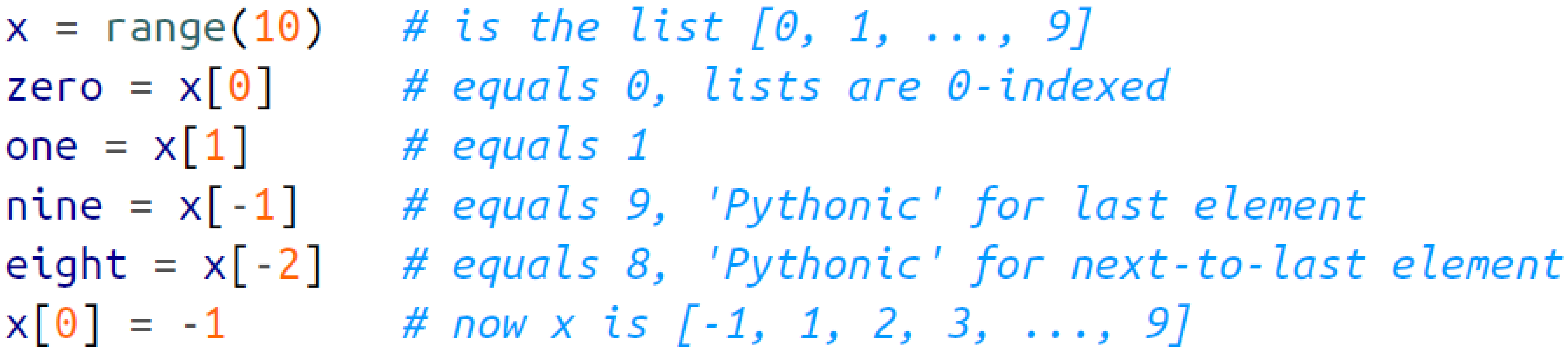
```
try:
    print 0 / 0
except ZeroDivisionError:
    print "cannot divide by zero"
```

Lists

- Most fundamental data structure



```
integer_list = [1, 2, 3]
heterogeneous_list = ["string", 0.1, True]
list_of_lists = [ integer_list, heterogeneous_list, [] ]

list_length = len(integer_list)     # equals 3
list_sum    = sum(integer_list)     # equals 6
```



```
x = range(10)   # is the list [0, 1, ..., 9]
zero = x[0]     # equals 0, lists are 0-indexed
one = x[1]      # equals 1
nine = x[-1]    # equals 9, 'Pythonic' for last element
eight = x[-2]   # equals 8, 'Pythonic' for next-to-last element
x[0] = -1       # now x is [-1, 1, 2, 3, ..., 9]
```

Lists

```
first_three   = x[:3]                       # [-1, 1, 2]
three_to_end = x[3:]                        # [3, 4, ..., 9]
one_to_four = x[1:5]                        # [1, 2, 3, 4]
last_three = x[-3:]                         # [7, 8, 9]
without_first_and_last = x[1:-1]            # [1, 2, ..., 8]
copy_of_x = x[:]                            # [-1, 1, 2, ..., 9]
```

- Python has an in operator to check for list membership

```
1 in [1, 2, 3]    # True
0 in [1, 2, 3]    # False
```

Lists

- Concatenate lists

```
x = [1, 2, 3]
x.extend([4, 5, 6])     # x is now [1,2,3,4,5,6]
```

- Don’t want to modify original list contents

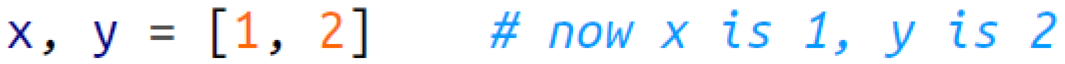
```
x = [1, 2, 3]
y = x + [4, 5, 6]       # y is [1, 2, 3, 4, 5, 6]; x is unchanged
```

- Append operation

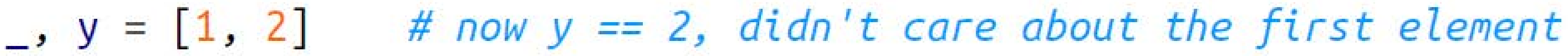
```
x = [1, 2, 3]
x.append(0)      # x is now [1, 2, 3, 0]
 y = x[-1]       # equals 0
 z = len(x)      # equals 4
```

Lists

- *Unpack* lists



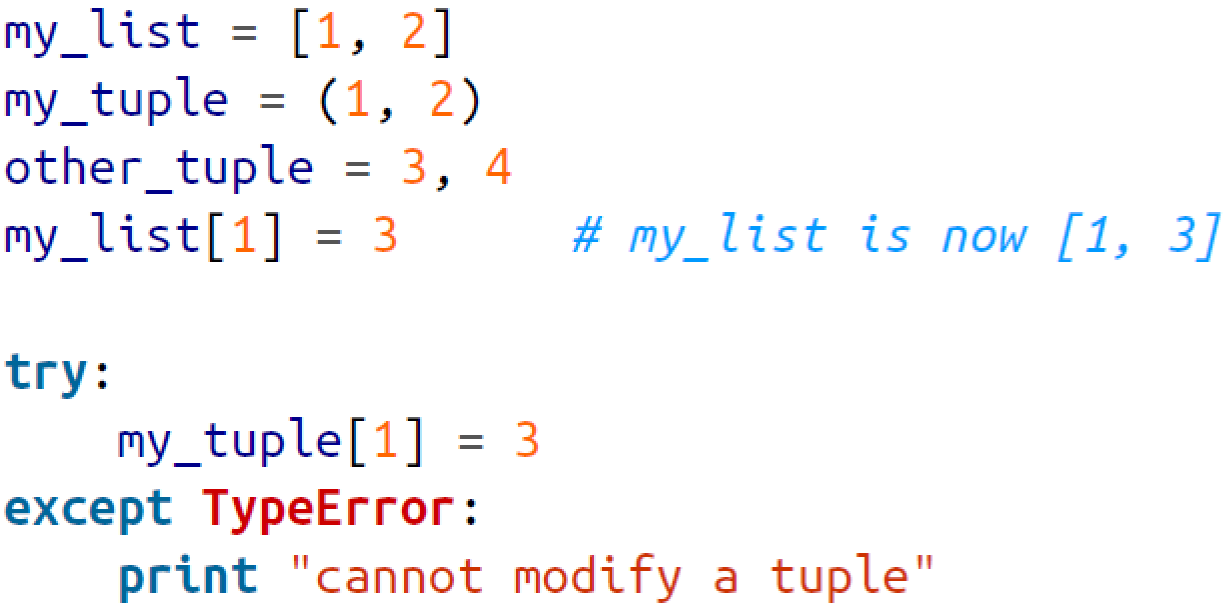
```
x, y = [1, 2]     # now x is 1, y is 2
```



```
_, y = [1, 2]     # now y == 2, didn't care about the first element
```

Tuples

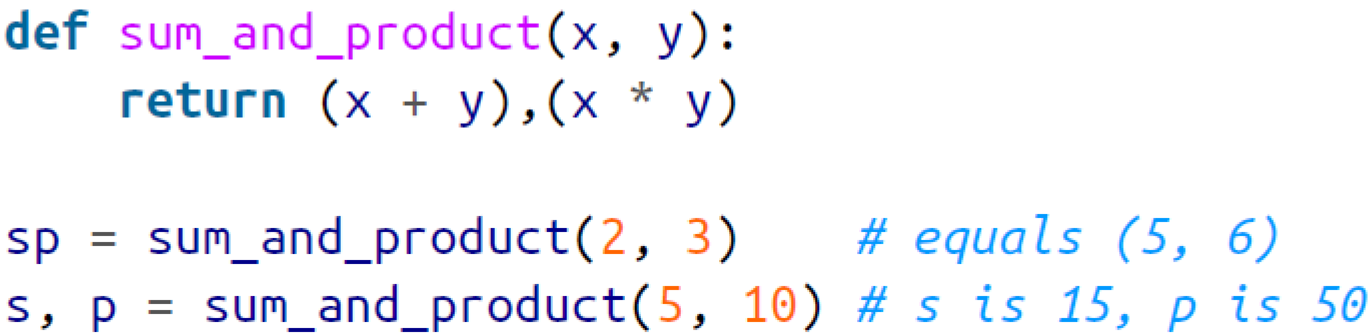
- Tuples are lists’ immutable cousins



```python
my_list = [1, 2]
my_tuple = (1, 2)
other_tuple = 3, 4
my_list[1] = 3      # my_list is now [1, 3]

try:
    my_tuple[1] = 3
except TypeError:
    print "cannot modify a tuple"
```

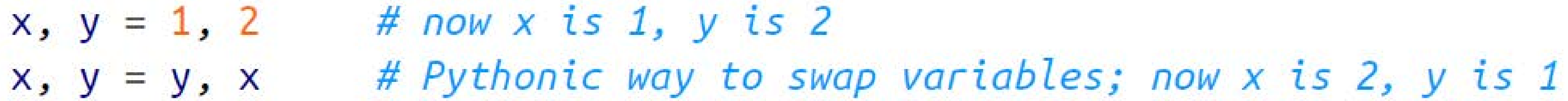
Tuples

- Tuples are a convenient way to return multiple values from functions



```
def sum_and_product(x, y):
    return (x + y),(x * y)

sp = sum_and_product(2, 3)    # equals (5, 6)
s, p = sum_and_product(5, 10) # s is 15, p is 50
```

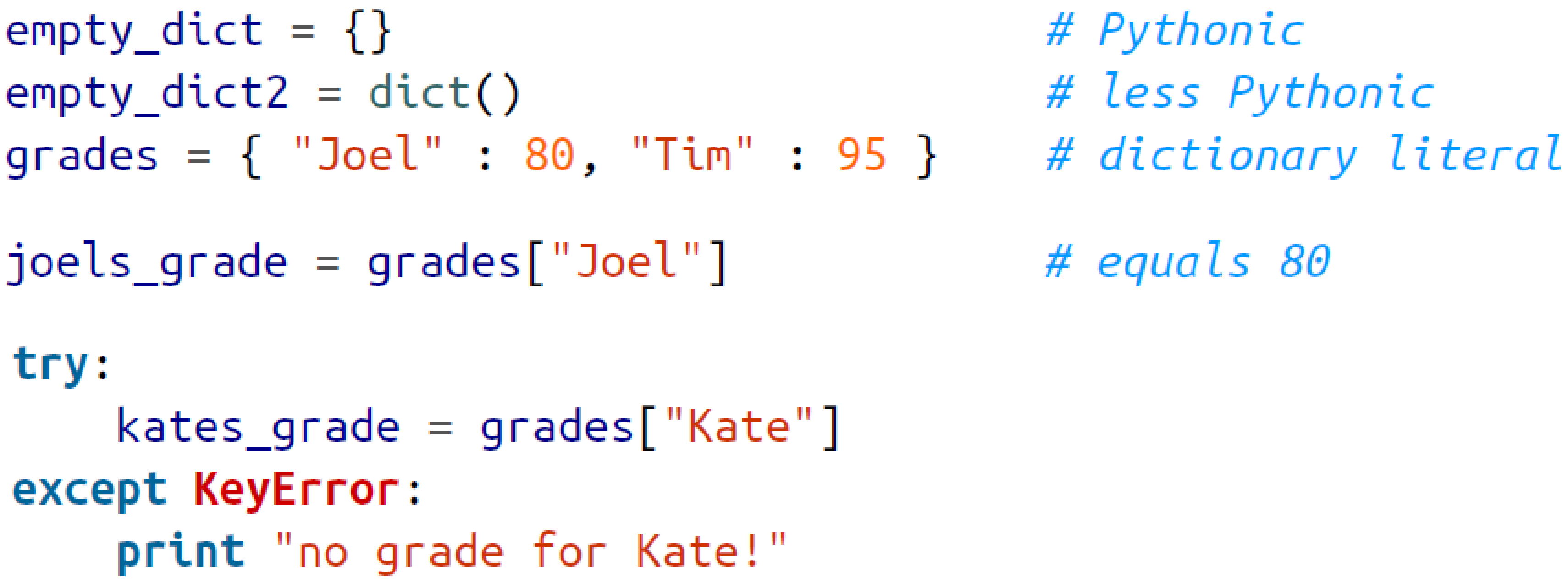
- Tuples (and lists) can also be used for *multiple assignment*:



```
x, y = 1, 2     # now x is 1, y is 2
x, y = y, x     # Pythonic way to swap variables; now x is 2, y is 1
```

Dictionaries

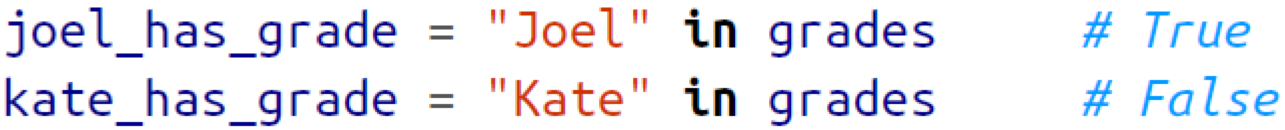
- Another fundamental data structure, which associates *values* with *keys*



```
empty_dict = {}                         # Pythonic
empty_dict2 = dict()                    # less Pythonic
grades = { "Joel" : 80, "Tim" : 95 }    # dictionary literal

joels_grade = grades["Joel"]            # equals 80

try:
    kates_grade = grades["Kate"]
except KeyError:
    print "no grade for Kate!"
```

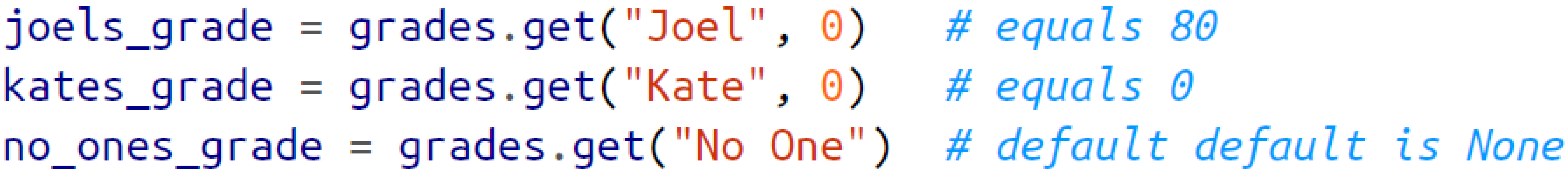
Dictionaries

- Check for the existence of a key



```
joel_has_grade = "Joel" in grades     # True
kate_has_grade = "Kate" in grades     # False
```

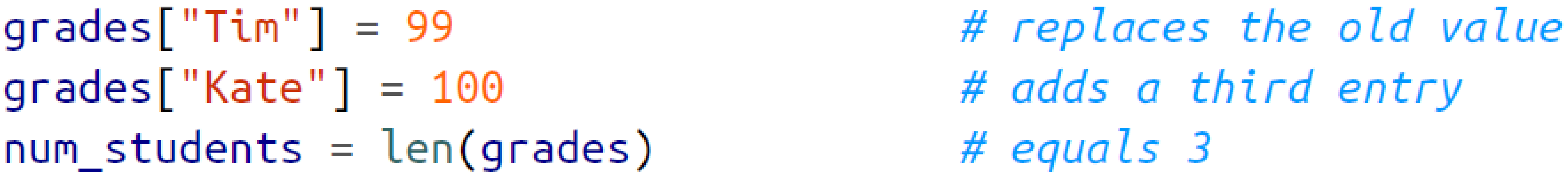
- Returns a default value (instead of raising an exception)



```
joels_grade = grades.get("Joel", 0)   # equals 80
kates_grade = grades.get("Kate", 0)   # equals 0
no_ones_grade = grades.get("No One")  # default default is None
```

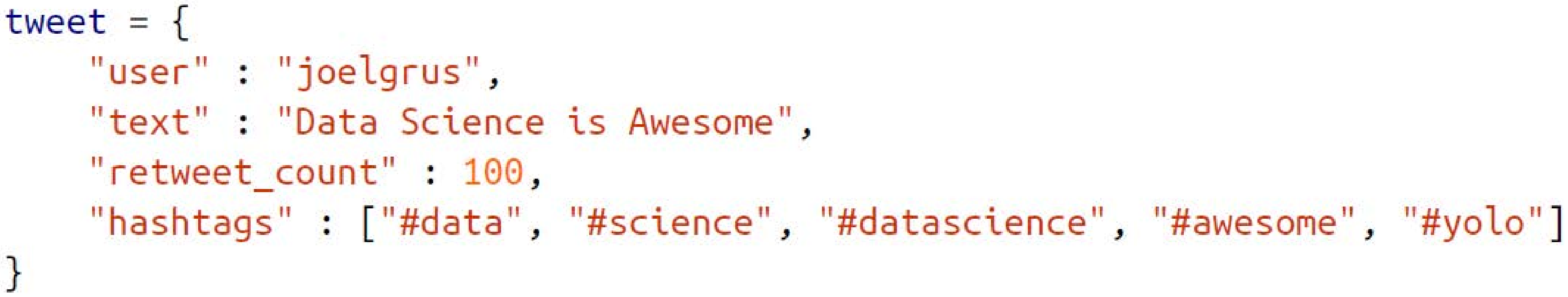
Dictionaries

- Assign key-value pairs using the same square brackets



```
grades["Tim"] = 99                    # replaces the old value
grades["Kate"] = 100                  # adds a third entry
num_students = len(grades)            # equals 3
```

- Use dictionaries as a simple way to represent structured data



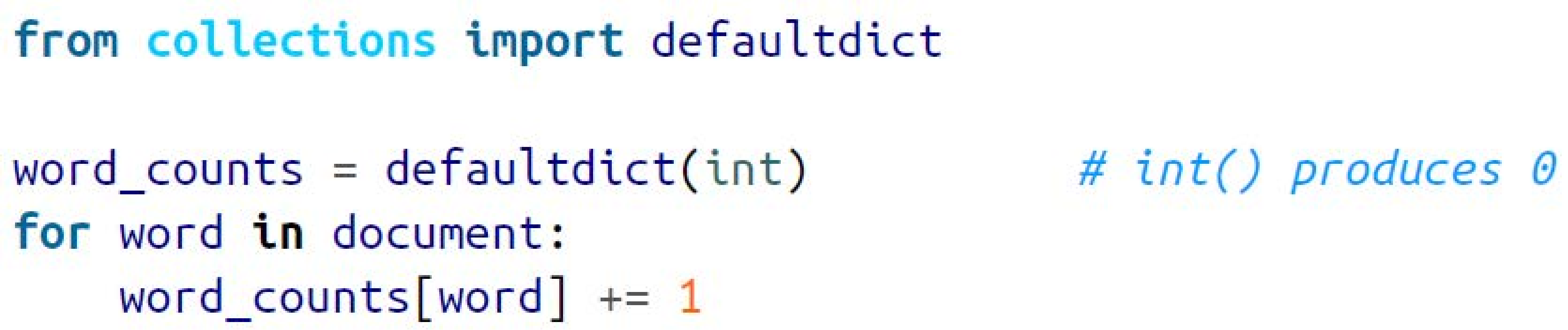
```
tweet = {
    "user" : "joelgrus",
    "text" : "Data Science is Awesome",
    "retweet_count" : 100,
    "hashtags" : ["#data", "#science", "#datascience", "#awesome", "#yolo"]
}
```

Dictionaries

```
tweet_keys   = tweet.keys()     # list of keys
tweet_values = tweet.values()   # list of values
tweet_items  = tweet.items()    # list of (key, value) tuples

"user" in tweet_keys            # True, but uses a slow list in
"user" in tweet                 # more Pythonic, uses faster dict in
"joelgrus" in tweet_values      # True
```

defaultdict

- A defaultdict is like a regular dictionary, except that when you try to look up a key it doesn’t contain, it first adds a value for it using a zero-argument function you provided when you created it



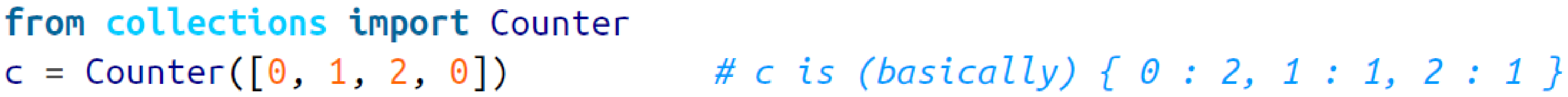
```
from collections import defaultdict

word_counts = defaultdict(int)          # int() produces 0
for word in document:
    word_counts[word] += 1
```

defaultdict

```
dd_list = defaultdict(list)             # list() produces an empty list
dd_list[2].append(1)                    # now dd_list contains {2: [1]}

dd_dict = defaultdict(dict)             # dict() produces an empty dict
dd_dict["Joel"]["City"] = "Seattle"     # { "Joel" : { "City" : Seattle"}}

dd_pair = defaultdict(lambda: [0, 0])
dd_pair[2][1] = 1                       # now dd_pair contains {2: [0,1]}
```

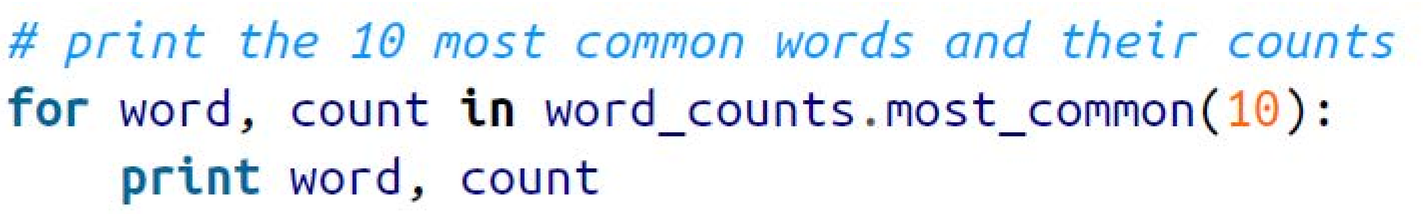
Counter

- A Counter turns a sequence of values into a defaultdict(int)-like object mapping keys to counts



```
from collections import Counter
c = Counter([0, 1, 2, 0])          # c is (basically) { 0 : 2, 1 : 1, 2 : 1 }
```

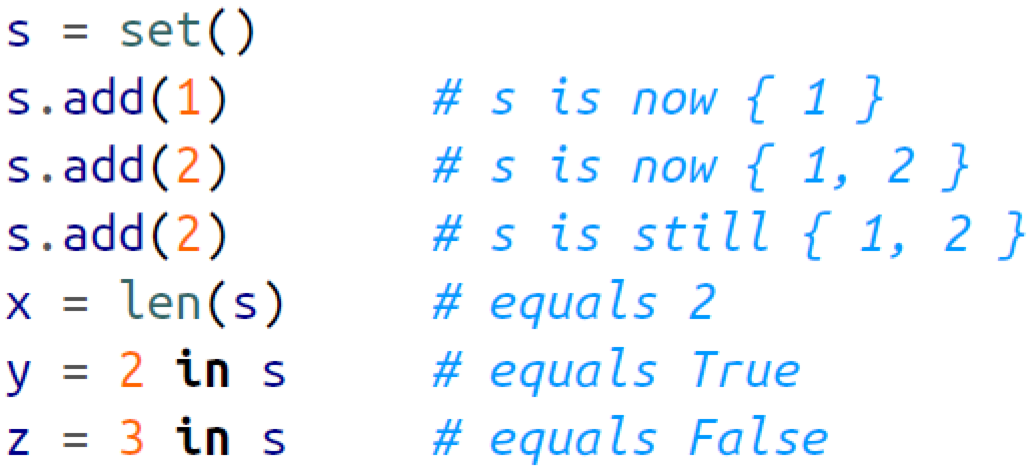
- most_common method



```
# print the 10 most common words and their counts
for word, count in word_counts.most_common(10):
    print word, count
```

Sets

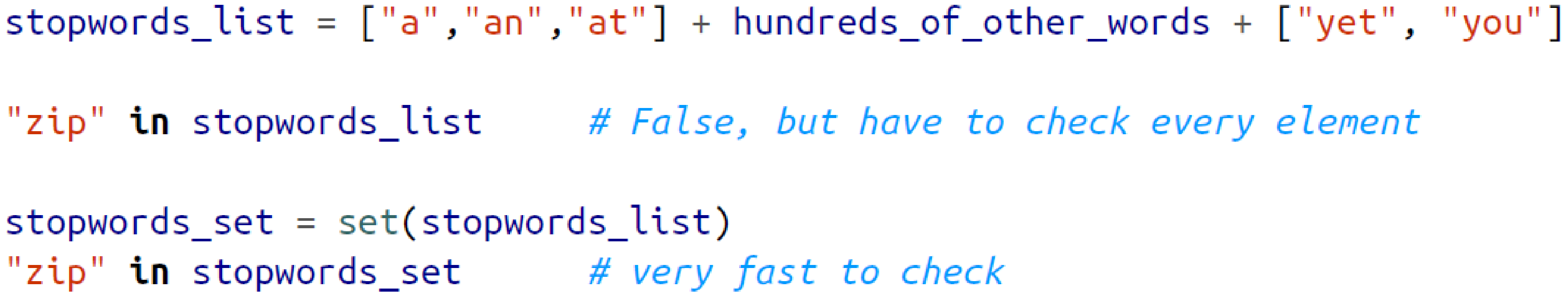
- Represents a collection of *distinct* elements



```
s = set()
s.add(1)        # s is now { 1 }
s.add(2)        # s is now { 1, 2 }
s.add(2)        # s is still { 1, 2 }
x = len(s)      # equals 2
y = 2 in s      # equals True
z = 3 in s      # equals False
```

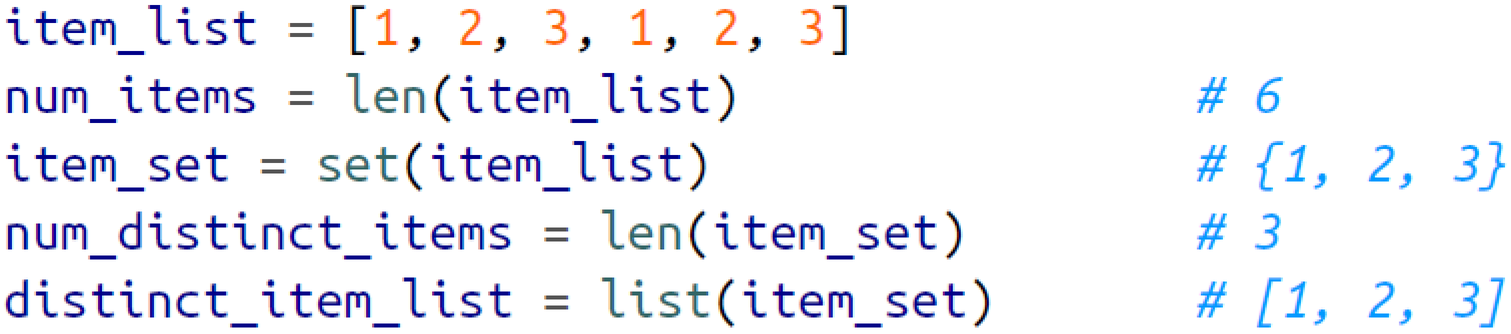
Sets

- in method is a very fast operation on sets



```
stopwords_list = ["a","an","at"] + hundreds_of_other_words + ["yet", "you"]

"zip" in stopwords_list      # False, but have to check every element

stopwords_set = set(stopwords_list)
"zip" in stopwords_set       # very fast to check
```

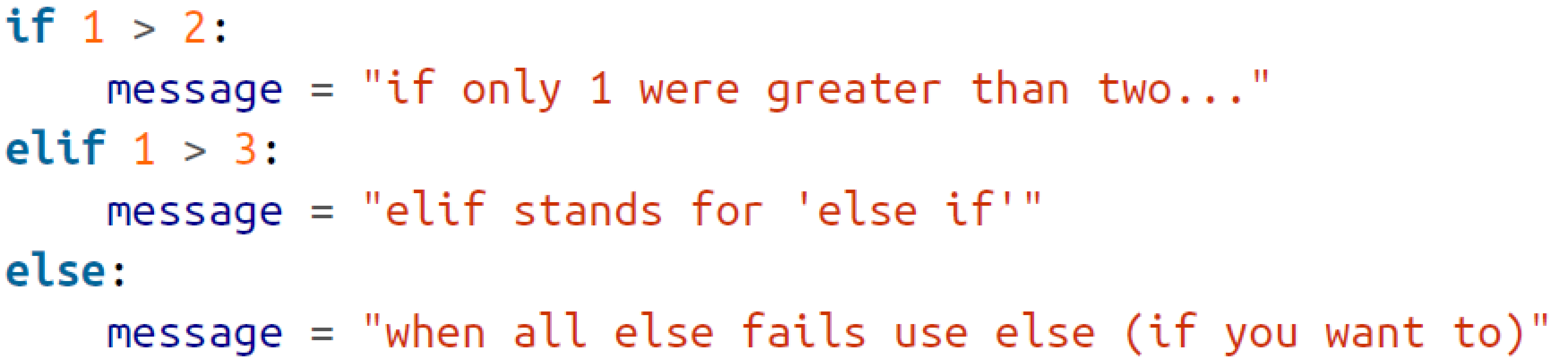
- To find the *distinct* items in a collection



```
item_list = [1, 2, 3, 1, 2, 3]
num_items = len(item_list)                    # 6
item_set = set(item_list)                     # {1, 2, 3}
num_distinct_items = len(item_set)            # 3
distinct_item_list = list(item_set)           # [1, 2, 3]
```

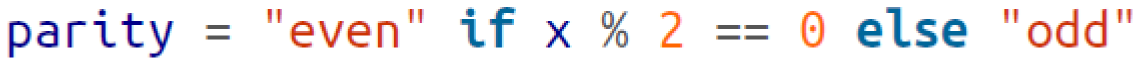
Control Flow

- if-else



```
if 1 > 2:
    message = "if only 1 were greater than two..."
elif 1 > 3:
    message = "elif stands for 'else if'"
else:
    message = "when all else fails use else (if you want to)"
```

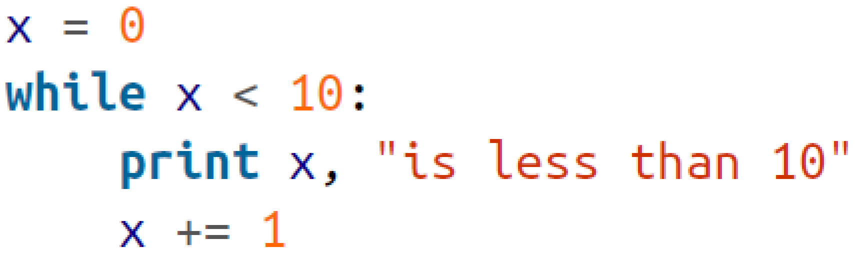
- *ternary* if-then-else



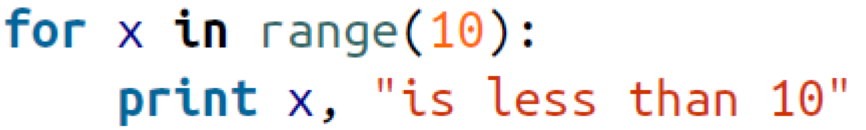
```
parity = "even" if x % 2 == 0 else "odd"
```

Control Flow

- while loop, or for-and-in

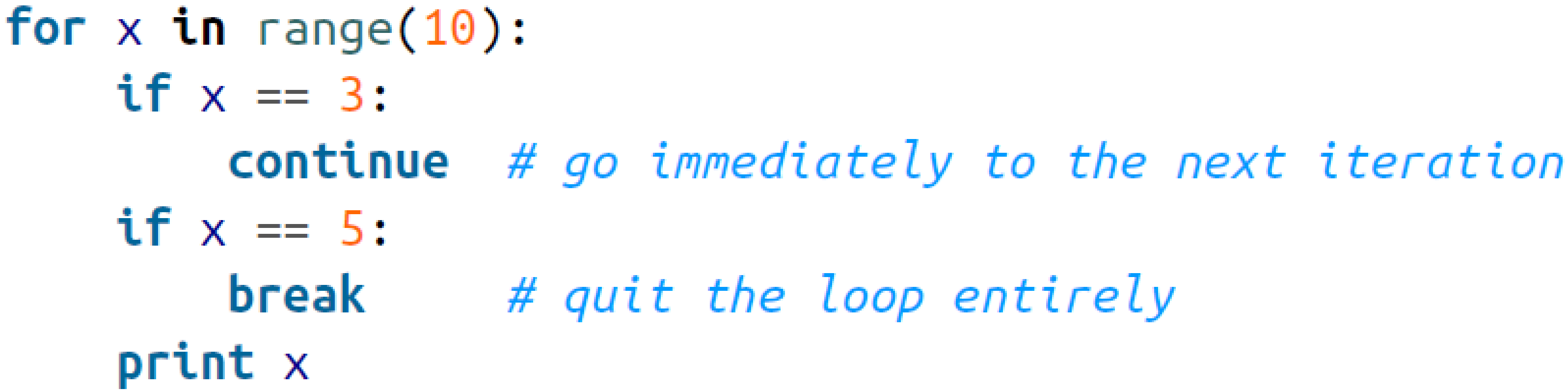


```
x = 0
while x < 10:
    print x, "is less than 10"
    x += 1
```



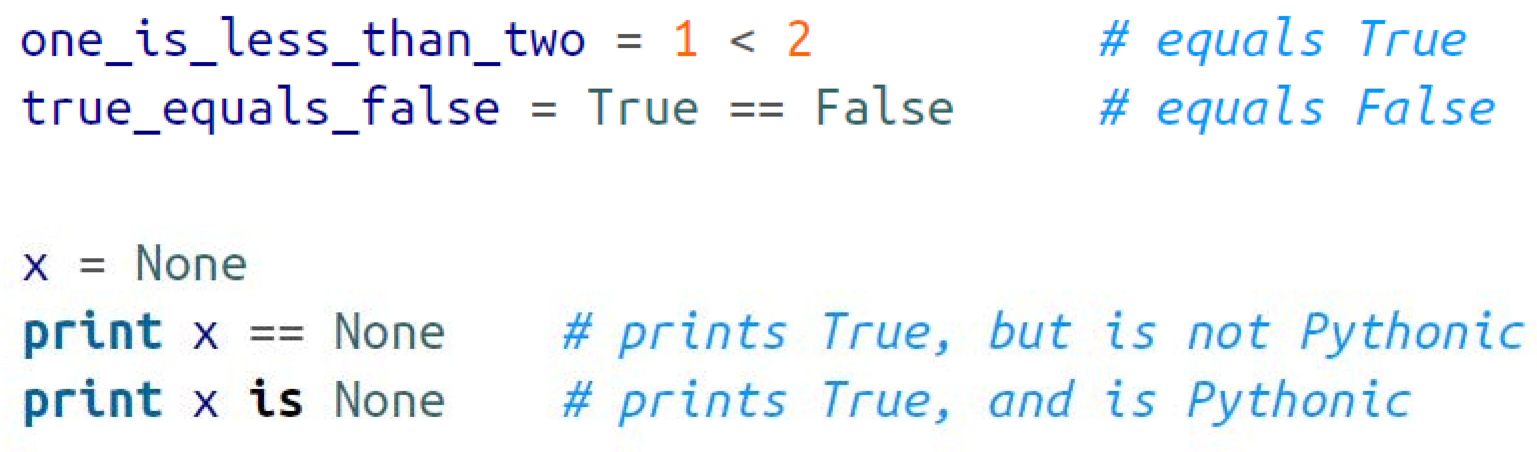
```
for x in range(10):
    print x, "is less than 10"
```

- continue and break



```
for x in range(10):
    if x == 3:
        continue  # go immediately to the next iteration
    if x == 5:
        break     # quit the loop entirely
    print x
```

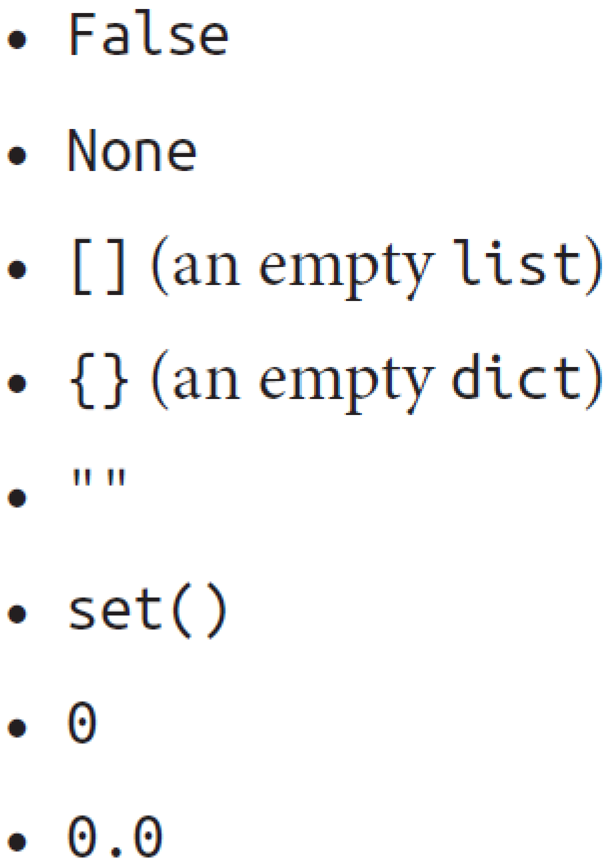
Truthiness/Booleans



```
one_is_less_than_two = 1 < 2          # equals True
true_equals_false = True == False     # equals False

x = None
print x == None    # prints True, but is not Pythonic
print x is None    # prints True, and is Pythonic
```

Truthiness/Booleans

- Falsy values



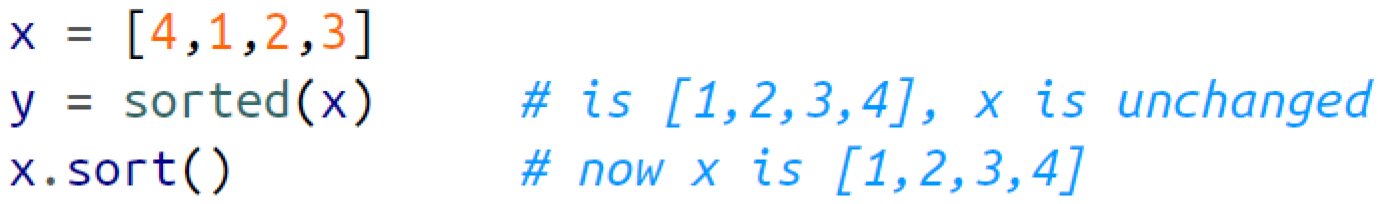
- `False`
- `None`
- `[]` (an empty list)
- `{}` (an empty dict)
- `""`
- `set()`
- `0`
- `0.0`

Outline

- The Basics
- The Not-So-Basics
- Object-oriented Programming

Sorting

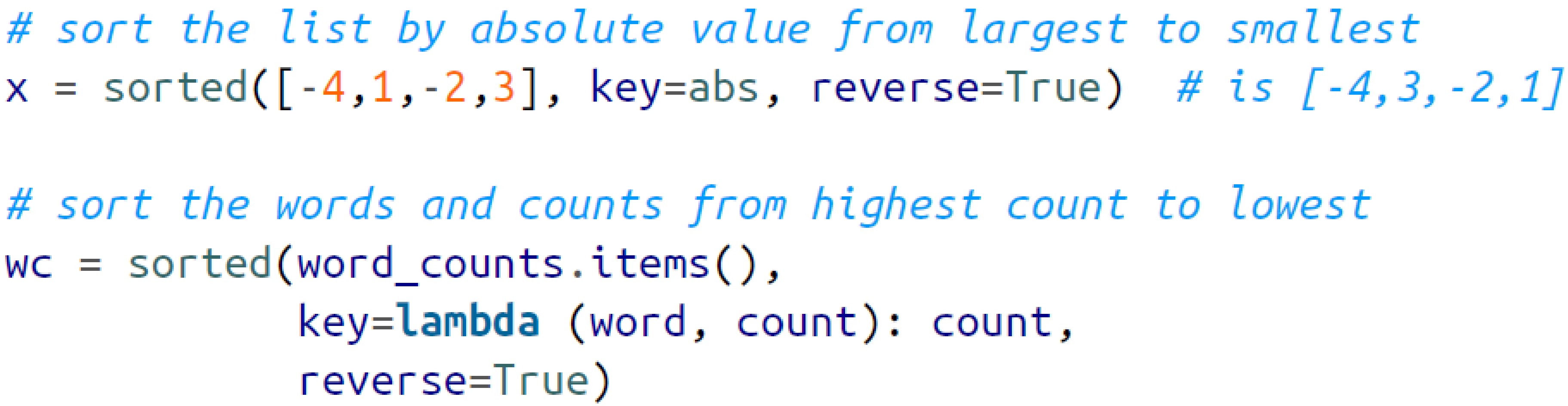
- Python list has a sort method that sorts it in place. If you don’t want to mess up your list, you can use the sorted function



```
x = [4,1,2,3]
y = sorted(x)     # is [1,2,3,4], x is unchanged
x.sort()          # now x is [1,2,3,4]
```

Sorting

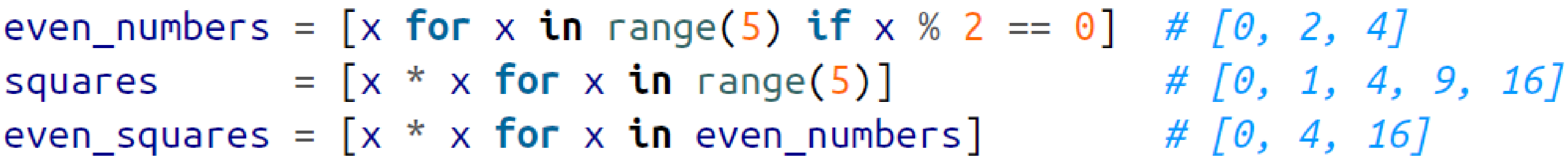
- Customized order and compare results of function



```
# sort the list by absolute value from largest to smallest
x = sorted([-4,1,-2,3], key=abs, reverse=True)  # is [-4,3,-2,1]

# sort the words and counts from highest count to lowest
wc = sorted(word_counts.items(),
            key=lambda (word, count): count,
            reverse=True)
```

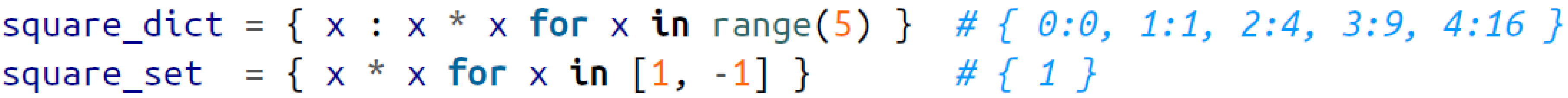
List Comprehensions

- Transform a list into another list by *list comprehensions*



```
even_numbers = [x for x in range(5) if x % 2 == 0]  # [0, 2, 4]
squares      = [x * x for x in range(5)]            # [0, 1, 4, 9, 16]
even_squares = [x * x for x in even_numbers]        # [0, 4, 16]
```

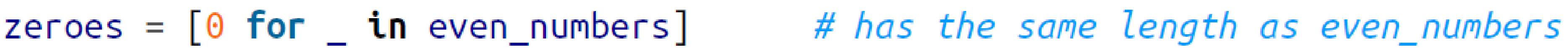
- Turn lists into dictionaries or sets



```
square_dict = { x : x * x for x in range(5) }  # { 0:0, 1:1, 2:4, 3:9, 4:16 }
square_set  = { x * x for x in [1, -1] }       # { 1 }
```

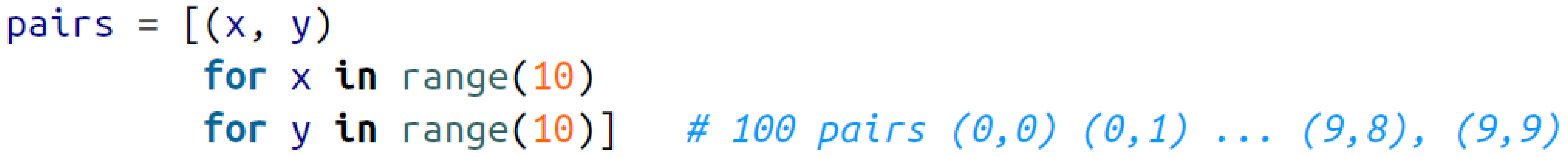
List Comprehensions

- Don’t need the value from the list



```python
zeroes = [0 for _ in even_numbers]      # has the same length as even_numbers
```

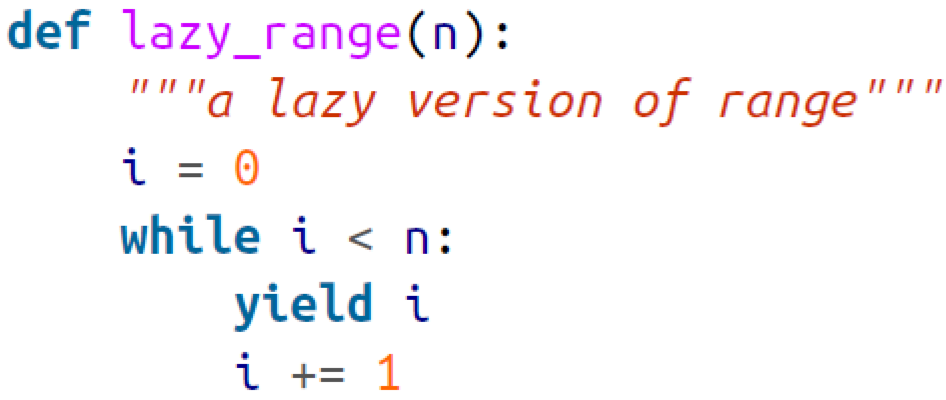
- Include multiple fors



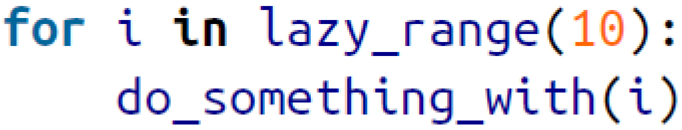
```python
pairs = [(x, y)
         for x in range(10)
         for y in range(10)]   # 100 pairs (0,0) (0,1) ... (9,8), (9,9)
```

Generators and Iterators

- A *generator* is something that you can iterate over (for us, usually using for) but whose values are produced only as needed (*lazily*)



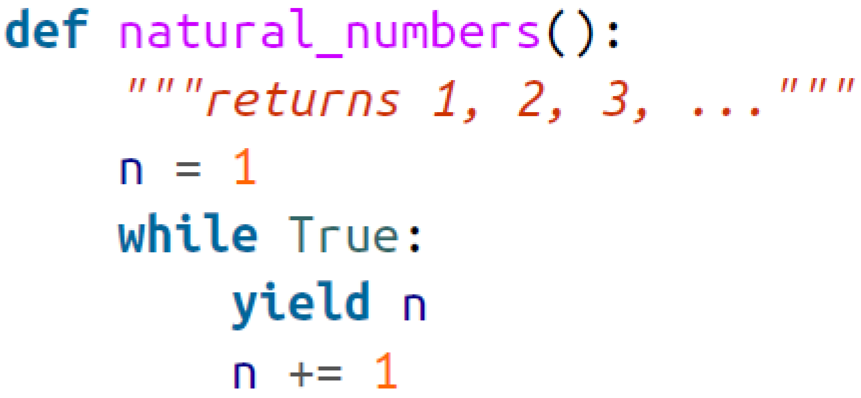
```
def lazy_range(n):
    """a lazy version of range"""
    i = 0
    while i < n:
        yield i
        i += 1
```



```
for i in lazy_range(10):
    do_something_with(i)
```

Generators and Iterators

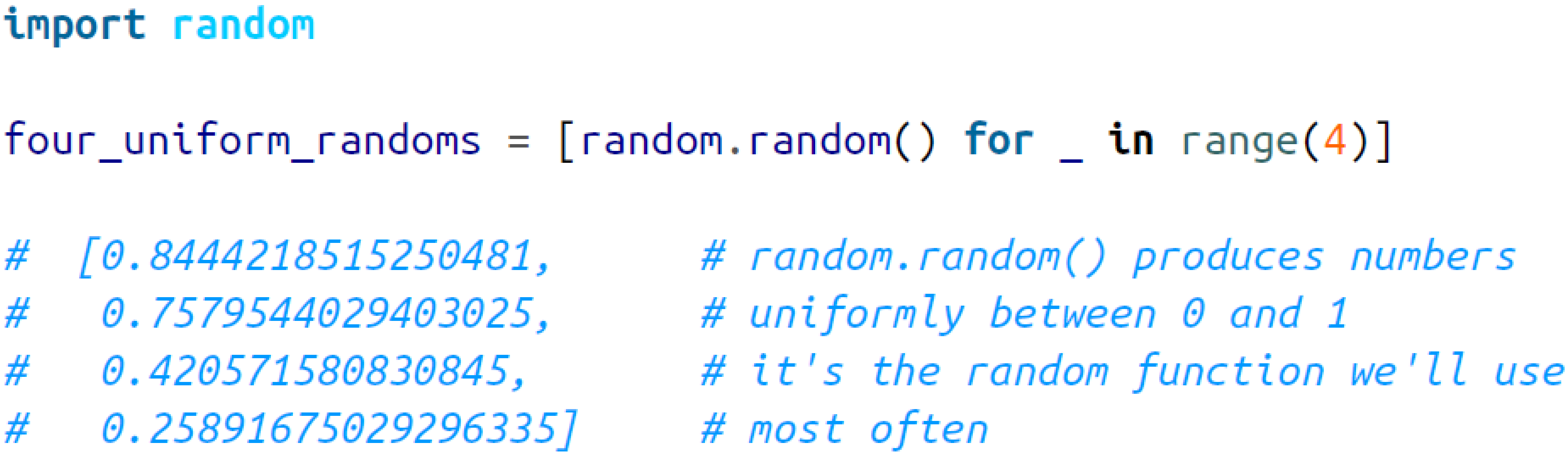
- You could even create an infinite sequence



```python
def natural_numbers():
    """returns 1, 2, 3, ..."""
    n = 1
    while True:
        yield n
        n += 1
```

Randomness

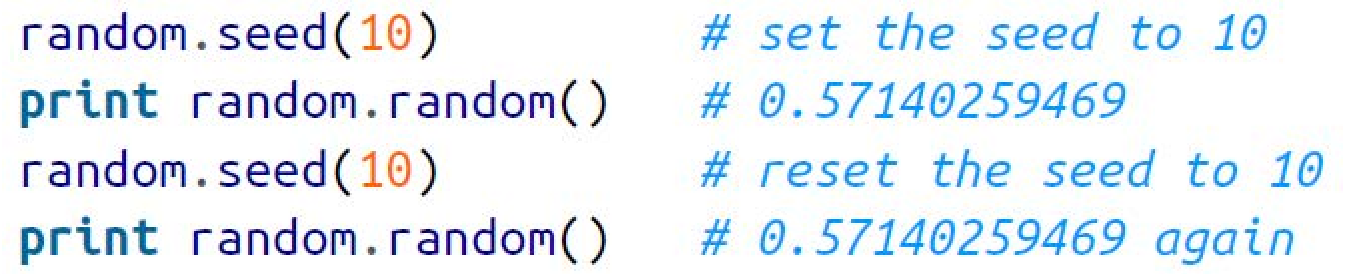
- Generate random numbers



```
import random

four_uniform_randoms = [random.random() for _ in range(4)]

#  [0.8444218515250481,      # random.random() produces numbers
#   0.7579544029403025,      # uniformly between 0 and 1
#   0.420571580830845,       # it's the random function we'll use
#   0.25891675029296335]     # most often
```

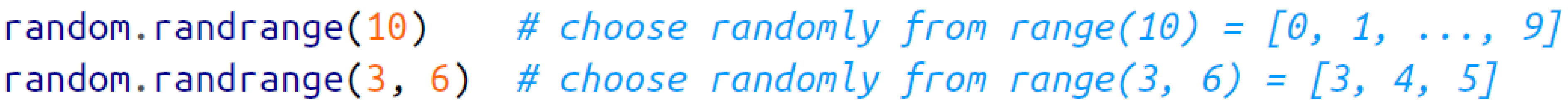
Randomness

- You can set with random.seed if you want to get reproducible results



```
random.seed(10)           # set the seed to 10
print random.random()     # 0.57140259469
random.seed(10)           # reset the seed to 10
print random.random()     # 0.57140259469 again
```

- Chosen randomly from the corresponding range()



```
random.randrange(10)     # choose randomly from range(10) = [0, 1, ..., 9]
random.randrange(3, 6)   # choose randomly from range(3, 6) = [3, 4, 5]
```

Randomness

- Randomly reorders the elements

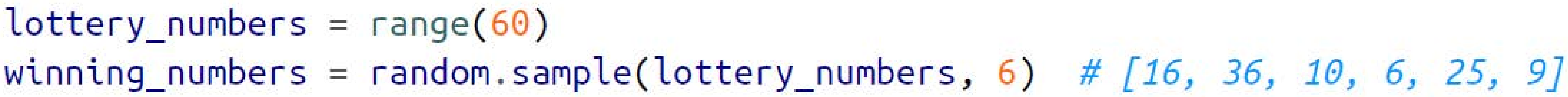
```
up_to_ten = range(10)
random.shuffle(up_to_ten)
print up_to_ten
# [2, 5, 1, 9, 7, 3, 8, 6, 4, 0]   (your results will probably be different)
```

- Randomly pick one element

```
my_best_friend = random.choice(["Alice", "Bob", "Charlie"])     # "Bob" for me
```

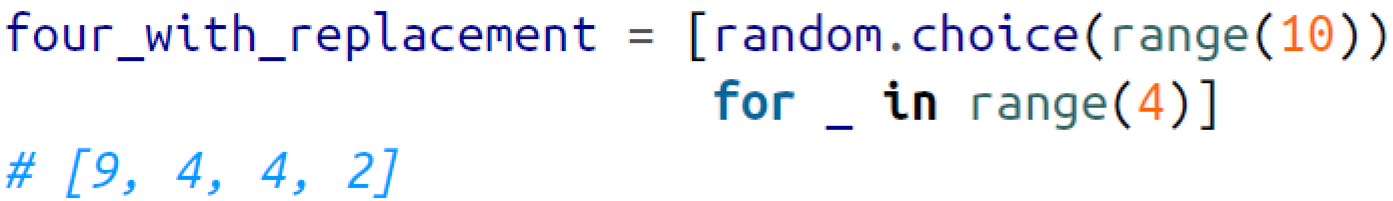
Randomness

- Randomly choose a sample of elements without replacement (no duplicates)



```
lottery_numbers = range(60)
winning_numbers = random.sample(lottery_numbers, 6)  # [16, 36, 10, 6, 25, 9]
```

- choose a sample of elements *with* replacement (allowing duplicates), just make multiple calls to random.choice



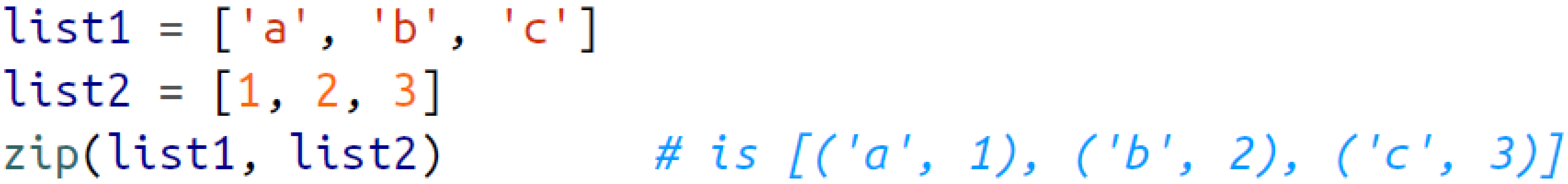
```
four_with_replacement = [random.choice(range(10))
                         for _ in range(4)]

# [9, 4, 4, 2]
```

Regular Expressions

```
import re

print all([                                 # all of these are true, because
    not re.match("a", "cat"),               # * 'cat' doesn't start with 'a'
    re.search("a", "cat"),                  # * 'cat' has an 'a' in it
    not re.search("c", "dog"),              # * 'dog' doesn't have a 'c' in it
    3 == len(re.split("[ab]", "carbs")),    # * split on a or b to ['c','r','s']
    "R-D-" == re.sub("[0-9]", "-", "R2D2") # * replace digits with dashes
    ])  # prints True
```

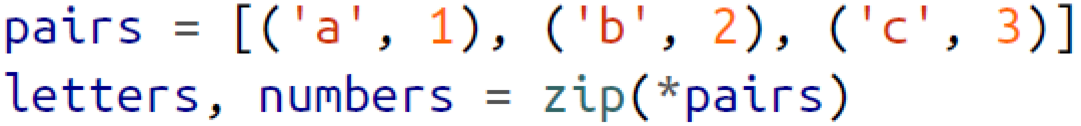
zip and Argument Unpacking

- zip transforms multiple lists into a single list of tuples of corresponding elements



```
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]
zip(list1, list2)        # is [('a', 1), ('b', 2), ('c', 3)]
```

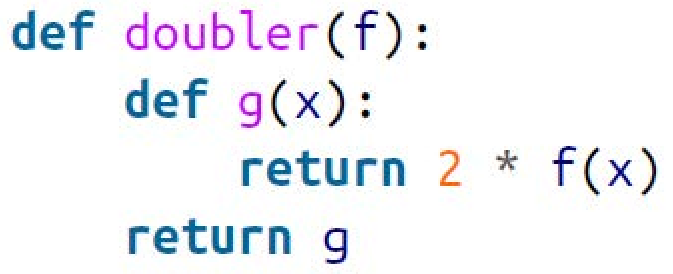
- “unzip” a list using a strange trick



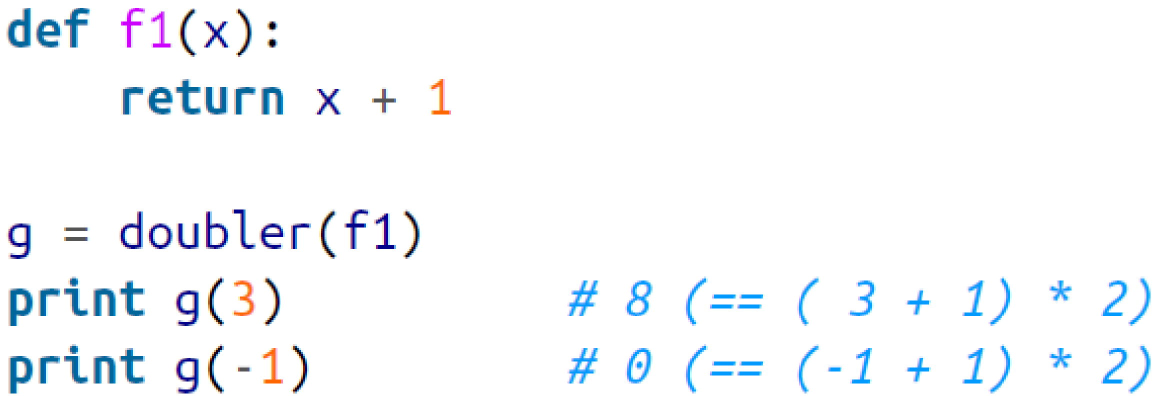
```
pairs = [('a', 1), ('b', 2), ('c', 3)]
letters, numbers = zip(*pairs)
```

args and kwargs

- Create a higher-order function that takes as input some function f and returns a new function



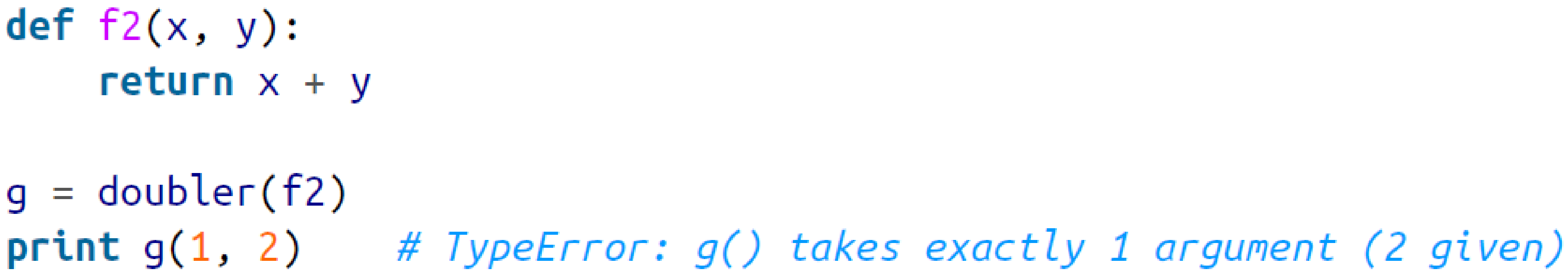
```
def doubler(f):
    def g(x):
        return 2 * f(x)
    return g
```



```
def f1(x):
    return x + 1

g = doubler(f1)
print g(3)            # 8 (== ( 3 + 1) * 2)
print g(-1)           # 0 (== (-1 + 1) * 2)
```

args and kwargs

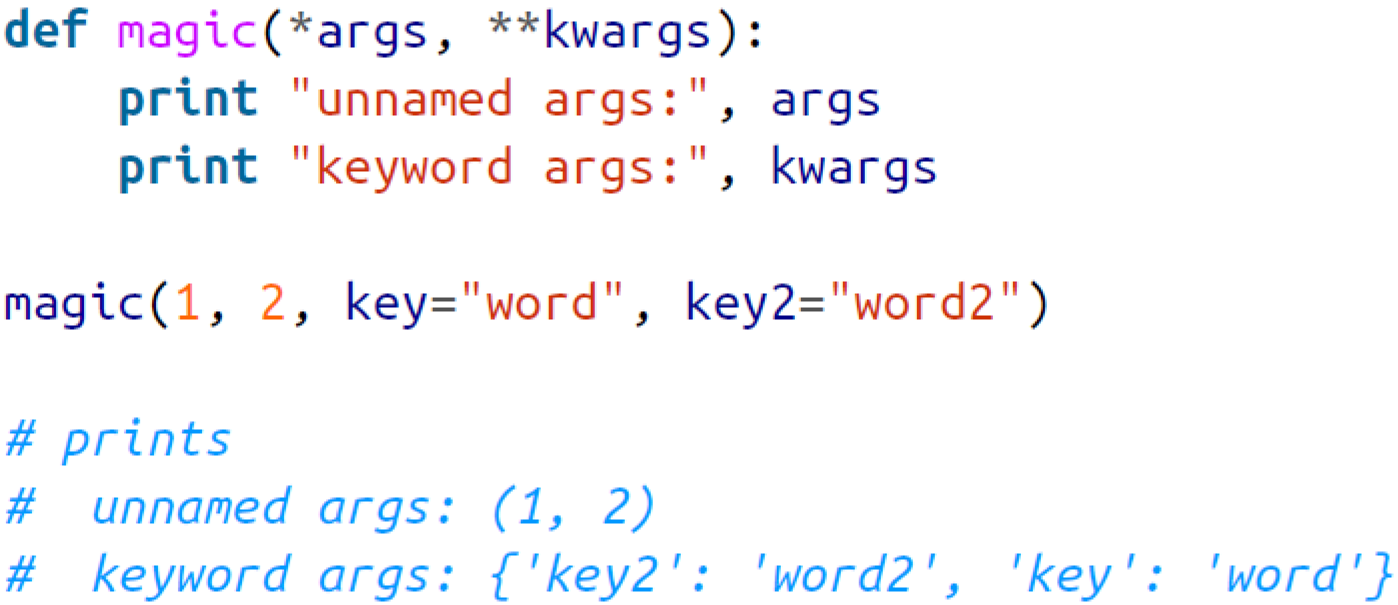
- … but breaks down with functions that take more than a single argument



```
def f2(x, y):
    return x + y

g = doubler(f2)
print g(1, 2)    # TypeError: g() takes exactly 1 argument (2 given)
```

args and kwargs

- Solution

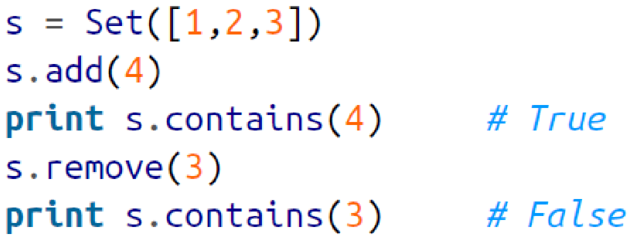


```
def magic(*args, **kwargs):
    print "unnamed args:", args
    print "keyword args:", kwargs

magic(1, 2, key="word", key2="word2")

# prints
#  unnamed args: (1, 2)
#  keyword args: {'key2': 'word2', 'key': 'word'}
```

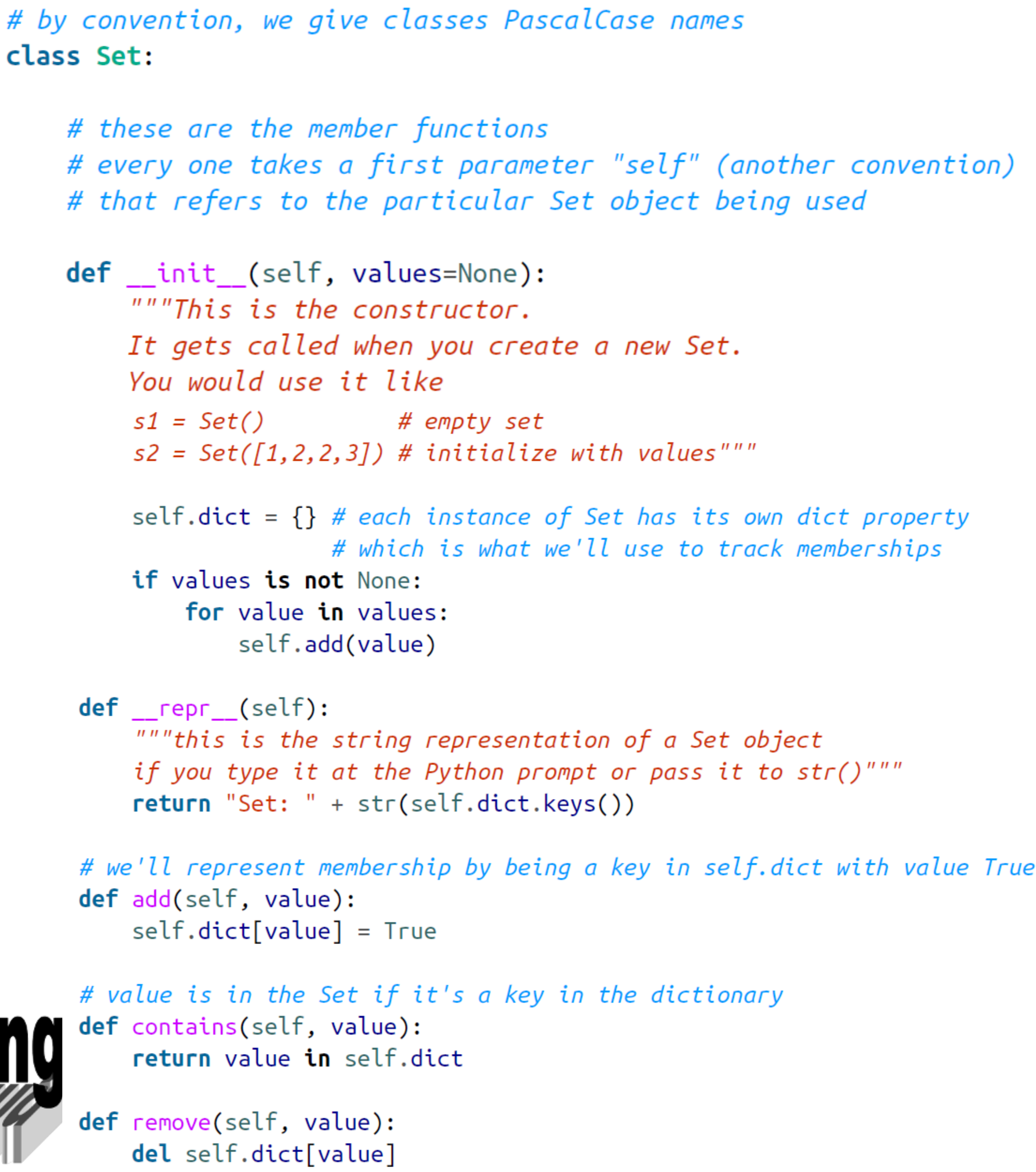
Outline

- The Basics
- The Not-So-Basics
- Object-oriented Programming

Object-oriented Programming



```
s = Set([1,2,3])
s.add(4)
print s.contains(4)      # True
s.remove(3)
print s.contains(3)      # False
```



```
# by convention, we give classes PascalCase names
class Set:

    # these are the member functions
    # every one takes a first parameter "self" (another convention)
    # that refers to the particular Set object being used

    def __init__(self, values=None):
        """This is the constructor.
        It gets called when you create a new Set.
        You would use it like
        s1 = Set()          # empty set
        s2 = Set([1,2,2,3]) # initialize with values"""

        self.dict = {} # each instance of Set has its own dict property
                       # which is what we'll use to track memberships
        if values is not None:
            for value in values:
                self.add(value)

    def __repr__(self):
        """this is the string representation of a Set object
        if you type it at the Python prompt or pass it to str()"""
        return "Set: " + str(self.dict.keys())

    # we'll represent membership by being a key in self.dict with value True
    def add(self, value):
        self.dict[value] = True

    # value is in the Set if it's a key in the dictionary
    def contains(self, value):
        return value in self.dict

    def remove(self, value):
        del self.dict[value]
```

Concluding Remarks

- 30-minute work!