Programming 2

Dries Decuyper
Serhat Erdogan
Janne Gilis
Bart Thumas

Lambdas
Sorting
Min – Max
Iterable

PBA Toegepaste Informatica

2023-2024

AGENDA

- Lambdas
- Sorting
- Min – Max
- Iterable

Lambdas

Sorting

Min – Max

Iterable

Lambdas

- What are lambdas?
- Syntax
- Examples
- Multiple parameters
- Power of lambda-functions

WHAT ARE LAMBDAS?

- “Anonymous functions”
- Not going through formal python steps
- Small piece of code only to use in specific location
- Only for very simple functions: limited to a single expression (python specific limitation)

SYNTAX

```
lambda arguments: expression
```

- Keyword lambda
- Number of arguments
- An expression to execute on the arguments

EXAMPLES

```
numbers = [1, 2, 3, 4, 5]
cubed = lambda x: x ** 3

for num in numbers:
    print(cubed(num))
'''
result:
1
8
27
64
125
'''
```

```
names = ['joe', 'mike', 'sarah']
uppercase = lambda s: s.upper()

for name in names:
    print(uppercase(name))
'''
result:
JOE
MIKE
SARAH
'''
```

MULTIPLE PARAMETERS

```python
sumOfThree = lambda x, y, z: x + y + z

print(sumOfThree(1,2,3))
# result: 6
```

POWER OF LAMBDA-FUNCTIONS

```
def somepower(n):
    return lambda a: a ** n

squared = somepower(2)
cubed = somepower(3)
supercubed = somepower(4)

print(squared(5))
print(cubed(5))
print(supercubed(5))
'''
result:
25
125
625
'''
```

- Not necessary to write a different function for each case
- Define functions ahead of time with only a 2-lined function with an embedded lambda
- Even though function called in different locations, lambda still needed in one location

EXERCISE

- Try the following exercises

- 04-lambdas

Lambdas

Sorting

Min – Max

Iterable

Sorting

- Difference between sort and sorted
- __lt__
- Sorting based on keys
- Sorting by two (or more) values
- attrgetter

DIFFERENCE BETWEEN SORT AND SORTED

- sort() function will modify the list it is called on
- Has no return value

```
vegetables = ['squash', 'pea', 'carrot', 'potato']

vegetables.sort()

print(vegetables)
# result: ['carrot', 'pea', 'potato', 'squash']
```

- The sorted() function will create a new list containing a sorted version of the list it is given as parameter
- Has a return value, must assign to variable

```
vegetables = ['squash', 'pea', 'carrot', 'potato']

sorted_vegetables = sorted(vegetables)

print(sorted_vegetables)
# result: ['carrot', 'pea', 'potato', 'squash']
```

__LT__

```
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade

    def __lt__(self, other):
        return self.grade < other.grade

    def __str__(self):
        return f'{self.name}: {self.grade}'

    def __repr__(self):
        return str(self)

students = [Student('Jan', 95), Student('Eve', 45), Student('Nicole', 87), Student('Will', 56)]

students.sort()

print(students)
# result: [Eve: 45, Will: 56, Nicole: 87, Jan: 95]
```

- By default, sorting will rely on the < operator
- Define the __lt__ (lower than) dunder method in your class

SORTING BASED ON KEYS

```
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade

    def __str__(self):
        return f'{self.name}: {self.grade}'

    def __repr__(self):
        return str(self)

students = [Student('Jan', 95), Student('Eve',
45), Student('Nicole', 87), Student('Will', 56)]

students.sort(key=lambda student: student.grade)

print(students)
# result: [Eve: 45, Will: 56, Nicole: 87, Jan: 95]
```

```
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade

    def __str__(self):
        return f'{self.name}: {self.grade}'

    def __repr__(self):
        return str(self)

students = [Student('Jan', 95), Student('Eve', 45),
Student('Nicole', 87), Student('Will', 56)]

sorted_students = sorted(students, key=lambda student:
student.grade)

print(sorted_students)
# result: [Eve: 45, Will: 56, Nicole: 87, Jan: 95]
```

- ‘key’ keyword parameter with lambda
- __lt__ dunder method not needed anymore

SORTING BY TWO (OR MORE) VALUES

```
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade

    ...

students = [Student('Jan', 95), Student('Eve', 45), Student('Nicole', 87),
Student('Will', 56), Student('May', 87)]

students.sort(key=lambda student: (student.grade, student.name))

print(students)
# result: [Eve: 45, Will: 56, May: 87, Nicole: 87, Jan: 95]
```

- So, what if you want to sort students with the same grade?
- First sort them on their grade, then on their name alphabetically
- We can rely on tuples in the lambda-function used as a key

ATTRGETTER

```python
from operator import attrgetter

class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade

    def __str__(self):
        return f'{self.name}: {self.grade}'

    def __repr__(self):
        return str(self)

students = [Student('Jan', 95), Student('Eve', 45), Student('Nicole', 87), Student('Will', 56), Student('May', 87)]

students.sort(key=attrgetter('grade'))

print(students)
# result: [Eve: 45, Will: 56, Nicole: 87, May: 87, Jan: 95]
```

- From operator import attrgetter
- Quicker and shorter version of writing lambda obj: obj.member all the time
- Attrgetter(‘grade’) is the same as lambda obj: obj.grade

EXERCISE

- Try the following exercises

- 05-sorting

Lambdas

Sorting

Min – Max

Iterable

Min - Max

- Maximum of numbers
- Maximum of field from class
- Object that has maximum of field
- Maximum based on keys

MAXIMUM OF NUMBERS

```
numbers =
[1,5,3,8,2,0,4]

print(max(numbers))
# result: 8
```

- max() function

MAXIMUM OF FIELD FROM CLASS

```python
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade

    def __str__(self):
        return f'{self.name}: {self.grade}'

    def __repr__(self):
        return str(self)

students = [Student('Jan', 95), Student('Eve', 45), Student('Nicole', 87), Student('Will',
56), Student('May', 87)]

print(max([student.grade for student in students]))
# result: 95
```

- Use list comprehension to go to a list of grades, a field from a class you want to get the max from

OBJECT THAT HAS MAXIMUM OF FIELD

```python
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade

    def __str__(self):
        return f'{self.name}: {self.grade}'

    def __repr__(self):
        return str(self)

students = [Student('Jan', 95), Student('Eve', 45), Student('Nicole', 87), Student('Will',
56), Student('May', 87), Student('Yu', 95)]

max_grade = max([student.grade for student in students])
best_student = [student for student in students if student.grade == max_grade]

print(best_student)
# result: [Jan: 95, Yu: 95]
```

- First look for the max grade, then the object with this grade

MAXIMUM BASED ON KEYS

```
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade

    def __str__(self):
        return f'{self.name}: {self.grade}'

    def __repr__(self):
        return str(self)

students = [Student('Jan', 95), Student('Eve', 45), Student('Nicole', 87), Student('Will',
56), Student('May', 87), Student('Yu', 95)]

best_student = max(students, key=lambda student: student.grade)

print(best_student)
# result: Jan: 95
```

- We rather not go twice through a list to find the object
- Just like sort, max also can take a “key”-keyword which is a lambda function
- **ONLY** first occurrence is returned!!!

EXERCISE

- Try the following exercises

- 06-min-max

Iterable

- Definition
- Functions
- “Unpacking” iterables
- Enumerating iterables

DEFINITION

- An iterable is any Python object capable of returning its members one at a time, permitting it to be iterated over in a for-loop
- “Under the hood”, an iterable is any Python object with an __iter__() method or with a __getitem__() method that implements Sequence semantics
- Lists, tuples, strings, dictionaries, sets, …

FUNCTIONS

- list, tuple, dict, set: construct a list, tuple, dictionary, or set, respectively, from the contents of an iterable
- sum: sum the contents of an iterable
- sorted: return a list of the sorted contents of an interable
- any: returns True and ends the iteration immediately if bool(item) was True for any item in the iterable
- all: returns True only if bool(item) was True for all items in the iterable
- max: return the largest value in an iterable
- min: return the smallest value in an iterable

FUNCTIONS

```python
print(list("I am a cow"))
# result: ['I', ' ', 'a', 'm', ' ', 'a', ' ', 'c', 'o', 'w']

print(sum([1, 2, 3]))
# result: 6

print(sorted("gheliabciou"))
# result: ['a', 'b', 'c', 'e', 'g', 'h', 'i', 'i', 'l', 'o', 'u']

print(any((0, None, [], 0)))
# result: False

print(all([1, (0, 1), True, "hi"]))
# result: True

print(max((5, 8, 9, 0)))
# result: 9

print(min("hello"))
# result: 'e'
```

“UNPACKING” ITERABLES

```python
numbers = [7, 9, 11]

x = numbers[0]
y = numbers[1]
z = numbers[2]

print(x,y,z)
# result: 7 9 11
```

```python
numbers = [7, 9, 11]

x, y, z = numbers

print(x,y,z)
# result: 7 9 11
```

- Python provides an extremely useful functionality, known as iterable unpacking, which allows us to write the simple, elegant code

“UNPACKING” ITERABLES

```
students = [('Jan', 95), ('Eve', 45), ('Nicole', 87)]

for name, grade in students:
    print(f'name: {name}')
    print(f'grade: {grade}')
'''
Result:
name: Jan
grade: 95
name: Eve
grade: 45
name: Nicole
grade: 87
'''
```

- Iterable unpacking is particularly useful in the context of performing for-loops over iterables-of-iterables

ENUMERATING ITERABLES

```
for entry in enumerate("abcd"):
    print(entry)
'''
result:
(0, 'a')
(1, 'b')
(2, 'c')
(3, 'd')
'''
```

- The built-in enumerate function allows us to iterate over an iterable, while keeping track of the iteration count

ENUMERATING ITERABLES

```
none_indices = []
iter_cnt = 0

for item in [2, None, -10, None, 4, 8]:
    if item is None:
        none_indices.append(iter_cnt)
    iter_cnt = iter_cnt + 1

print(none_indices)
# result: [1,3]
```

```
none_indices = []

for iter_cnt, item in enumerate([2, None, -10, None, 4, 8]):
    if item is None:
        none_indices.append(iter_cnt)

print(none_indices)
# result: [1, 3]
```

- In general, the enumerate function accepts an iterable as an input, and returns a new iterable that produces a tuple of the iteration-count and the corresponding item from the original iterable. Thus the items in the iterable are being enumerated

EXERCISE

- Try the following exercises
- 07-iterable