Bootcamp 134 | Python

Course 06 | Basic Python

Amir Hossein Chegouniyan

Head of the Technical Team at Dariche Tejarat

Lecturer of Python – Django at Maktab Sharif

Amirhossein-chegounian

Content

Functional Programming

- Immutability:
  - Cannot change the value of inputs.
- Pure functions:
  - Depends only on inputs.
- Avoiding side effects:
  - In programming, a side effect occurs when a function does something beyond calculating an output.
- Benefits

Lambda

- In Python, a lambda is a way to create anonymous functions — functions without a name.
- Lambda functions are usually used for short, simple operations, especially when defining a full function using def would be too long or unnecessary.
  - *lambda arguments: expression*

Lambda | Example

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

add = lambda x, y: x + y
print(add(3, 5))  # Output: 8
```

Map

- Applying a function to all items in a list or iterable
  - *map(function, list) # return a list*
- For example:

```
numbers = [1, 2, 3, 4, 5]
Squ = list(map(lambda x: x * x, numbers))
```

Filter

- Filtering items in an iterable based on a condition
  - *filter(function, list) # return a list*
  - *Function return a Boolean value*
- For example:

```
numbers = [1, 2, 3, 4, 5]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
```

Reduce

- For accumulating results (e.g., summing elements). Good for when you want to convert all elements to a single value.
  - *reduce(function, list) # return a single value*
  - *Function has two args*
- For example:

```
from functools import reduce
numbers = [1, 2, 3, 4, 5]
total = reduce(lambda x, y: x + y, numbers)
```

Generators

- Generator is a function that return a iterator
  - *Generator function: use from yield*
  - *Generator expression: (expression for i in <iterator> condition (optional))*

Generators | Function Example

```
def count_up_to(n):
    i = 1
    while i <= n:
        yield i       # use from yield
        i += 1

for num in count_up_to(5):
    print(num)
```

Generators | Expression Example

```
squares = (x**2 for x in range(1, 6))

for s in squares:
    print(s)
```

- What is the difference between list comprehensions and generators?

Decorator

- A decorator is a function that takes another function or class, extends or modifies its behavior without changing the original code.
- Is a very powerful and useful tool in python.
- What is the difference between a function and a decorator?

Decorator | Syntax and Use

```
def <name_of_decorator>(func):
        def wrapper():
                <every code before execute func>
                func()
                <every code after  execute func>
        return wrapper

@<name_of_decorator>
def <any_function>():
        <body of function>
```

Recursion

- Using a function within itself
- For example:

```
number = [1,2,3,4,5]
def new_func(inp):
        print(inp[-1])
        inp.pop()
        if len(inp) > 0:
                new_func(inp)
new_func(number)
```

Sets | Intro

- Are unordered, unchangeable, and do not allow duplicates
- Type: <class ‘set’>

Sets | Create

<name_of_set> = { 'value_01', 'value_02'}

- *For example:*

  student_codes = {"102548", "102487", "103479", "103241"}

- Can not access to set members.
- For access to members of set, use from loop and in/not in keyword

Sets | Add

```
<name_of_set>.add(<new value>)                # add new value to set
<name_of_set>.update(<iterator like list>)    # add members of iterator to set (not update)
```

- *For example:*

```
students_code.add("104987")
students_code.update(["104597", "104852"])
```

Sets | Delete

```
<name_of_set>.remove(<value>)       # error if not exist
<name_of_set>.discard(<value>)      # don't error if not exist
<name_of_set>.pop()                 # remove a random item
<name_of_set>.clear()               # empties the set
del <name_of_set>                   # remove the set completely
```

Sets | In Loops

```
for i in <name_of_set>:        # for waliking on set members
```

- *For example:*

```
for i, j in ali_info.items():
        print(f"{i}: {j}")
```

Sets | Operators

```
set3 = set1.union(set2)              # union two set and don’t change origin
set3 = set1 | set2                   # union two set and don’t change origin
set1.update(set2)                    # union two set and change origin
set3 = set1.intersection(set2)       # intersect two set and don’t change origin
set3 = set1 & set2                   # intersect two set and don’t change origin
set1.intersection_update(set2)       # intersect two set and change origin
```

Sets | Operators 2

```
set3 = set1.difference(set2)          # difference two set and don't change origin
set3 = set1 - set2                    # difference two set and don't change origin
set1.difference_update(set2)          # difference two set and change origin
set3 = set1.symmetric_difference(set2)         # two-sided difference two set and don't change origin
set1. symmetric_ difference_update(set2)       # two-sided difference two set and change origin
```

Itertools Module

- The itertools library in Python is one of the standard modules designed to work with iterators.

Itertools Module | Count

```
from itertools import count

for i in count(10, 2):
    print(i)
    if i > 20:
        break
```

```
10
12
14
16
18
20
22
```

Itertools Module | Chain

```
from itertools import chain

a = [1, 2, 3]
b = ['a', 'b']
result = list(chain(a, b))
print(result)
```

```
[1, 2, 3, 'a', 'b']
```

Itertools Module | Cycle

```
from itertools import cycle

count = 0
for item in cycle(['A', 'B', 'C']):
    print(item)
    count += 1
    if count == 6:
        break
```

```
A
B
C
A
B
C
```

Itertools Module | Combinations

```
from itertools import combinations

items = ['A', 'B', 'C']
for combo in combinations(items, 2):
        print(combo)
```

```
('A', 'B')
('A', 'C')
('B', 'C')
```

Itertools Module | Permutations

```
from itertools import permutations

items = ['A', 'B', 'C']
for perm in permutations(items, 2):
    print(perm)
```

```
('A', 'B')
('A', 'C')
('B', 'A')
('B', 'C')
('C', 'A')
('C', 'B')
```

Any question?

Next course

- Introduction to Version Control
- Git Basics
- Creating a Local Repository
- Working with GitHub
- Basic GitHub Workflow
- Group Practice