python-control_flow-iterations-functions

October 14, 2020

1 Python for Data Science

Control Flow, Iterations, Functions, Classes

2 Control Flow

- Without control flow, programs are sequences of statements
- With control flow you execute code
- **conditionally** (`if`, `else`)
- **repeatedly** (`for`, `while`)

2.1 Conditional Statements: `if-elif-else`:

[1]:

```python
x = 1e6

if x == 0:
    print(x, "is zero")
elif x > 0:
    print(x, "is positive")
elif x < 0:
    print(x, "is negative")
else:
    print(x, "is unlike anything I've ever seen...")
```

```
1000000.0 is positive
```

2.2 `for` loops

- Iterate over each element of a collection
- Python makes this look like almost natural language:

```
for [each] value in [the] list
```

[2]:

```python
for N in [2, 3, 5, 7]:
    print(N, end=' ') # print all on same line
```

```
2 3 5 7
```

[3]:

```python
for N in range(5):
    print(N, end=' ') # print all on same line
```

```
0 1 2 3 4
```

2.3 while loops

Iterate until a condition is met

[4]:

```python
i = 0
while i < 10:
    print(i, end=' ')
    i += 1
```

```
0 1 2 3 4 5 6 7 8 9
```

3 Functions

Remember the print statement

```python
print('abc')
```

`print` is a function and `'abc'` is an argument.

[5]:

```python
# multiple input arguments
print('abc','d','e','f','g')
```

```
abc d e f g
```

[6]:

```python
# keyword arguments
print('abc','d','e','f','g', sep='--')
```

```
abc--d--e--f--g
```

3.1 Defining Functions

[7]:

```python
def add(a, b):
    """
    This function adds two numbers

    Input
    a: a number
    b: another number

    Returns sum of a and b
    """
    result = a + b
    return result
```

[8]:

```
add(1,1)
```

[8]: 2

[9]:

```
def add_and_print(a, b, print_result):
    """
    This function adds two numbers

    Input
    a: a number
    b: another number
    print_result: boolean, set to true if you'd like the result printed

    Returns sum of a and b
    """
    result = a + b
    if print_result:
        print("Your result is {}".format(result))
    return result
```

[10]:

```
add_and_print(1, 1, True)
```

```
Your result is 2
```

[10]: 2

3.2 Default Arguments

[11]:

```
def add_and_print(a, b, print_result=True):
    """
    This function adds two numbers

    Input
    a: a number
    b: another number
    print_result: boolean, set to true if you'd like the result printed

    Returns sum of a and b
    """
    result = a + b
    if print_result:
        print("Your result is {}".format(result))
    return result
```

[12]:
```
add_and_print(1, 1)
```

```
Your result is 2
```

[12]:
```
2
```

3.3 *args and **kwargs: Flexible Arguments

[13]:
```
def add_and_print(*args, **kwargs):
    """
    This function adds two numbers

    Input
    a: a number
    b: another number
    print_result: boolean, set to true if you'd like the result printed

    Returns sum of a and b
    """
    result = 0
    for number in args:
        result += number
    if 'print_result' in kwargs.keys() and kwargs['print_result']:
        print("Your result is {}".format(result))
    return result
```

[14]:
```
add_and_print(1, 1, 1, print_result=True, unknown_argument='ignored')
```

```
Your result is 3
```

[14]:
```
3
```

[15]:
```
list_of_numbers = [1,2,3,42-6]
add_and_print(*list_of_numbers)
```

[15]:
```
42
```

3.4 Anonymous (lambda) Functions

[16]:

```python
add = lambda x, y: x + y
add(1, 2)
```

[16]:

```
3
```

4 Classes

- Python is an object oriented language
- Classes provide a means of bundling data and functionality together
- Classes allow for inheriting functionality

[17]:

```python
class Person:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def is_adult(self):
        return self.age > 18
```

[18]:

```python
p1 = Person("John", 36)

print(p1.name)
print(p1.age)
print(p1.is_adult())
```

```
John
36
True
```

[19]:

```python
class Student(Person):
    """A class inheriting fields and methods from class Person"""

p2 = Student("Peter", 20)
p2.is_adult()
```

[19]:

```
True
```

4.1 Some Convenient Special Functions

- Printing a String representation of an object: `__repr__`
- For calling an object: `__call__`
- Many more for specialized objects like iterables (just create an object and type `.__ + <TAB>`)

4.1.1 Nice String Representations of Objects with __repr__

[20]:

```
# the string representation of the Person class is not very informative
p1
```

[20]:

```
<__main__.Person at 0x10813cac0>
```

[21]:

```
# defining a __repr__ function that returns a string can help
class PrintableStudent(Student):
    def __repr__(self):
        return f"A student with name {self.name} and age {self.age}"

p3 = PrintableStudent("Michael Mustermann", 25)
p3
```

[21]:

```
A student with name Michael Mustermann and age 25
```

4.1.2 Clean APIs using __call__ for obvious usages of Objects

[22]:

```
# defining a __call__ function can help to keep APIs simple
class CallableStudent(PrintableStudent):
    def __call__(self, other_student):
        print(f"{self.name} calls {other_student.name}")

p4 = CallableStudent("Michael Mustermann", 25)
p4(p2)
```

```
Michael Mustermann calls Peter
```

5 List Comprehensions

A simple way of compressing a list building for loop into single statement

[23]:

```
L = []
for n in range(12):
    L.append(n ** 2)
L
```

[23]:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

[24]:

```
[n ** 2 for n in range(12)]
```

[24]:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

5.1 Conditional List Comprehensions

Including an `if` statement in list comprehensions

```
[25]: [n ** 2 for n in range(12) if n % 3 == 0]
```

```
[25]: [0, 9, 36, 81]
```

6 Set Comprehensions

Same as for lists, but for sets

```
[26]: {n**2 for n in range(12)}
```

```
[26]: {0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121}
```

7 Dict Comprehensions

Same as for lists, but for dictionaries

```
[27]: {n:n**2 for n in range(12)}
```

```
[27]: {0: 0,
       1: 1,
       2: 4,
       3: 9,
       4: 16,
       5: 25,
       6: 36,
       7: 49,
       8: 64,
       9: 81,
       10: 100,
       11: 121}
```

8 Generator Comprehensions

Generators generate values one by one. More on this later.

```
[28]: (n**2 for n in range(12))
```

```
[28]: <generator object <genexpr> at 0x1081b2190>
```

[29]:

```
# generators can be turned into lists
list((n**2 for n in range(12)))
```

[29]:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

9 Iterators

- An object over which Python can iterate are called `Iterators`
- Iterators
  - have a `__next__` method that returns the next element
  - have an `__iter__` method that returns self
- The builtin function `iter` turns any iterable in an iterator

[30]:

```
my_iterator = iter([1,2])
print(next(my_iterator))
print(next(my_iterator))
print(next(my_iterator))
```

```
1
2


      ␣
↪---------------------------------------------------------------------------

       StopIteration                             Traceback (most recent call␣
↪last)

       <ipython-input-30-bdcbfa3d0082> in <module>
         2 print(next(my_iterator))
         3 print(next(my_iterator))
   ----> 4 print(next(my_iterator))


       StopIteration:
```

9.1 Custom Iterators

[31]:

```
class Squares(object):

    def __init__(self, start, stop):
       self.start = start
       self.stop = stop
    def __iter__(self): return self

    def __next__(self):
        if self.start >= self.stop:
            raise StopIteration
        current = self.start * self.start
        self.start += 1
        return current

iterator = Squares(1, 5)
[i for i in iterator]
```

```
[31]: [1, 4, 9, 16]
```

9.2 Useful Builtin Iterators

9.2.1 enumerate

Often you need not only the elements of a collection but also their index

[32]:

```
L = [2, 4, 6]
for i in range(len(L)):
    print(i, L[i])
```

```
0 2
1 4
2 6
```

Instead you can write

[33]:

```
L = [2, 4, 6]
for idx, element in enumerate(L):
    print(idx, element)
```

```
0 2
1 4
2 6
```

9.2.2 zip

Zips together two iterators

[34]:

```
L = [2, 4, 6, 8, 10]
R = [3, 5, 7, 9, 11]
for l, r in zip(L, R):
    print(l, r)
```

```
2 3
4 5
6 7
8 9
10 11
```

9.2.3 Unzipping with `zip`

An iterable of tuples can be unzipped with `zip`, too:

[35]:

```
zipped = [('a',1), ('b',2)]
letters, numbers = zip(*zipped)
print(letters)
print(numbers)
```

```
('a', 'b')
(1, 2)
```

9.2.4 `map`

Applies a function to a collection

[36]:

```
def power_of(x, y=2):
    return x**2

for n in map(power_of, range(5)):
    print(n)
```

```
0
1
4
9
16
```

9.2.5 `filter`

Filters elements from a collection

[37]:

```
def is_even(x):
    return x % 2 == 0

for n in filter(is_even, map(power_of, range(5))):
    print(n)
```

```
0
4
16
```

[38]:

```
# compressing the above for loop
print(*filter(is_even, map(power_of, range(5))))
```

```
0 4 16
```

9.3 Specialized Iterators: itertools

9.3.1 Permutations

Iterating over all permutations of a list

[39]:

```
from itertools import permutations
my_iterator = range(3)
p = permutations(my_iterator)
print(*p)
```

```
(0, 1, 2) (0, 2, 1) (1, 0, 2) (1, 2, 0) (2, 0, 1) (2, 1, 0)
```

9.3.2 Combinations

Iterating over all unique combinations of N values within a list

[40]:

```
from itertools import combinations
c = combinations(range(4), 2)
print(*c)
```

```
(0, 1) (0, 2) (0, 3) (1, 2) (1, 3) (2, 3)
```

9.3.3 Product

Iterating over all combinations of elements in two or more iterables

[41]:

```
from itertools import product
my_iterator = range(3)
another_iterator = iter(['a', 'b'])
yet_another_iterator = iter([True, False])
p = product(my_iterator, another_iterator, yet_another_iterator)
print(*p)
```

```
(0, 'a', True) (0, 'a', False) (0, 'b', True) (0, 'b', False) (1, 'a', True) (1,
'a', False) (1, 'b', True) (1, 'b', False) (2, 'a', True) (2, 'a', False) (2,
'b', True) (2, 'b', False)
```

9.3.4 Chaining

Use Case: Chaining multiple iterators allows to combine file iterators

[42]:
```
from itertools import chain
my_iterator = range(3)
another_iterator = iter(['a', 'b'])
yet_another_iterator = iter([True, False])
p = chain(my_iterator, another_iterator, yet_another_iterator)
print(*p)
```

```
0 1 2 a b True False
```

9.3.5 Chaining for Flattening

Turning a nested collection like `[['a','b'],'c']` into a flat one like `['a','b','c']` is called **flattening**

[43]:
```
from itertools import chain
my_nested_list = [['a','b'],'c']
p = chain(*my_nested_list)
print(*p)
```

```
a b c
```

10 Generators - A Special Kind of Iterator

Generators make creation of iterators simpler.

Generators are built by calling a function that has one or more `yield` expression

[44]:
```
def squares(start, stop):
    for i in range(start, stop):
        yield i * i

generator = squares(1, 10)
[i for i in generator]
```

[44]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

11 When to use Iterators vs Generators

- Every Generator is an Iterator - but not vice versa
- Generator implementations can be simpler: `python generator = (i*i for i in range(a, b))`

- Iterators can have rich state