Python 05

Elias Wachmann

2024

Content

1. for - Loops
2. while - Loops
3. break & continue
4. String formatting
5. List Comprehensions
6. Bonus: Bitwise Operations

for - Loops

For loop

A `for` loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string). The `in` keyword is used to iterate over the sequence in a for loop.

```
1 numbers = [1, 2, 3, 4, 5]
2 for number in numbers:
3     print(number, end=", ")
4 # Output: 1, 2, 3, 4, 5,
```

For loop - range

`range` is often used in for loops to iterate over a sequence of numbers.

```
1 for i in range(2, 15, 3):
2     print(i, end=", ")
3 # Output: 2, 5, 8, 11, 14,
4
5 for i in range(5):
6     print(i, end=", ")
7 # Output: 0, 1, 2, 3, 4,
```

For loop - enumerate

`enumerate` is often used in for loops to iterate over a sequence and have an automatic counter.

```
1 names = ['Alice', 'Bob', 'Charlie']
2 for number, name in enumerate(names):
3     print(number, name, end=", ")
4     # Output: 0 Alice, 1 Bob, 2 Charlie,
```

For loop - zip

`zip` is often used in for loops to iterate over two or more sequences at the same time.

```
for number, name in enumerate(names):
    print(f"{name} is {age[number]} old", end=", ")
    # Output: Alice is 22 old, Bob is 42 old, Charlie is 16 old, Diana is 50 old,
```

results in same output as:

```
for name, age in zip(names, age):
    print(f"{name} is {age} old", end=", ")
    # same output as above
```

while - Loops

While loop

A `while` loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

```
1 count = 0
2 while count < 10:
3     # end line with ", " instead of "\n"
4     print(count, end=", ")
5     count += 1  # same as count = count + 1
```

While loop

Loop is executed repeatedly until `count` is not smaller than 10 anymore.

```
1 count = 0
2 while count < 10:
3     # end line with ", " instead of "\n"
4     print(count, end=", ")
5     count += 1  # same as count = count + 1
```

Output: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9,`

While loop

```
1  def add(a, b):
2      return a + b
3
4
5  a, b = 0, 1
6  # multiple conditions can also be used
7  while a < 5 or add(a, b) < 15:
8      print(a, b, end=", ")
9      # same as a = b * 2 and b = a + b
10     a, b = b * 2, a + b
11 # Output: 0 1, 2 1, 2 3, 6 5
```

break & continue

break & continue

break can stop loops earlier.

```
1  i = 0
2  while True:  # infinite loop
3      if i == 5:
4          # but we can break out of it
5          break
6      i += 1
7
8  for i in range(10):
9      if i == 7:
10         # we can also break out of for loops
11         break
12     print(i)
```

break & continue

Search for a specific value and break if found:

```
1  fruits = ["apple", "banana", "cherry"]
2  search_item = "banana"
3  i = 0
4
5  while i < len(fruits):
6      if fruits[i] == search_item:
7          print("Found", search_item, "at
       index", i)
8          break
9      i += 1
10 else:
11     print(search_item, "not found in list")
```

break & continue

Same example with a for loop:

```
fruits = ["apple", "banana", "cherry"]
search_item = "banana"

for fruit in fruits:
    if fruit == search_item:
        print(search_item, "found in list")
        break
else:
    print(search_item, "not found in list")
```

break & continue

continue can be used to prematurely end the current iteration of a loop and continue with the next iteration.

```
1 for i in range(1, 11):
2     if i % 2 == 0:
3         continue
4     print(i, end=", ")
```

Output:

```
1, 3, 5, 7, 9,
```

String formatting

Good to know: formatting in `print()`

```
1 name = "John"
2 age = 30
3 print("I'm {} and I am {} years old.".format
    (name, age))
4 print(f"I'm {name} and I am {age} years old.
    ")
5 print("I'm " + name + " and I am " + str(age
    ) + " years old.")
```

Same output for all three examples:

```
I'm John and I am 30 years old.
```

Good to know: formatting in `print()`

Format floats with a certain number of decimals places:

```
1 x = 42.42
2 y = 3.15
3 print(f"{x:.2f} + {y:.2f} = {(x+y):.2f}")
4 # a different way to do the same thing
5 print("{:.2f} + {:.2f} = {:.2f}".format(x, y
      , x + y))
6 # another way
7 print("%0.2f + %0.2f = %0.2f" % (x, y, x + y
     ))
8 # Output: 42.42 + 3.15 = 45.57
```

List Comprehensions

List Comprehensions

List comprehensions are a concise and efficient way to create a new list by applying an expression to each element of an existing list, optionally filtered by a conditional statement.
They provide a more readable and Pythonic alternative to traditional for loops and if statements.

List Comprehensions - Syntax

Basic syntax of a list comprehension:

```
[expression for item in list]
```

`expression` can be any valid expression, like a function call or a mathematical operation.

```
1 my_list = [x for x in range(1, 5)]
2 print(my_list)  # [1, 2, 3, 4]
3 double_list = [2 * x for x in my_list]
4 print(double_list)  # [2, 4, 6, 8]
```

List Comprehensions - Syntax

Syntax list comprehension with conditional statement:

```
[expression for item in list if conditional]
```

`conditional` statements can be used to decide if a given item should be included in the created list

```
1 my_list = [x for x in range(1, 5)]
2 print(my_list)  # [1, 2, 3, 4]
3 odds = [x for x in my_list if x % 2 != 0]
4 print(odds)  # [1, 3]
```

List Comprehensions - Syntax

Syntax list comprehension with nested for loops:

```
[expression for item1 in list1 for item2 in
                                    list2]
```

```
1 list1 = ['red', 'green', 'blue']
2 list2 = ['square', 'circle']
3 combinations = [f"{x} {y}!" for x in list1
    for y in list2]
4 # Combinations: ['red square!', 'red circle
    !', 'green square!', 'green circle!', '
    blue square!', 'blue circle!']
```

List Comprehensions - Examples

```
# Using list comprehension
original_list = [1, 2, 3, 4, 5, 6, 7, 8, 9,
    10]
even = [x for x in original_list if x % 2 ==
    0]
print(even)  # Output: [2, 4, 6, 8, 10]
# Using for loop and if statements
even = []
for x in original_list:
    if x % 2 == 0:
        even.append(x)
print(even)  # Output: [2, 4, 6, 8, 10]
```

more examples

Bonus: Bitwise Operations

Bitwise Operations - Basics

Bitwise Operators are used to compare (binary) numbers. They are used to perform bit by bit operation.
For example, 2 is 10 in binary and 7 is 111. If we perform a binary AND on 2 and 7 we get:

	0010
AND	0111
	0010

Each bit is compared and if both bits are 1, the result is 1, otherwise the result is 0.

Bitwise Operations - Tables

Bitwise AND ($\wedge$), OR ($\vee$), XOR ($\oplus$) and NOT ($\neg$) behave as shown in the following tables:

A	B	A $\wedge$ B	A $\vee$ B	A $\oplus$ B	$\neg$ A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Bitwise operations perform a check on each bit of the number. If the result of the truth table is 1, the bit is set to 1, otherwise it is set to 0.

Bitwise Operations - Examples

```
      0101
AND   1100
-----------
      0100
```

```
      0101
OR    1100
-----------
      1101
```

```
      0101
XOR   1100
-----------
      1001
```

```
      1100
NOT   ----
      0011
```

Bitwise Operations - Shifts

Bitwise shifts are used to shift the bits of a number to the left or right. Using variables `sl` and `sr` we get:

	`sl`	`sl` $\ll 1$	`sl` $\ll 2$	`sl` $\ll 3$
Binary number	1	10	100	1000
Decimal number	1	2	4	8

	`sr`	`sr` $\gg 1$	`sr` $\gg 2$	`sr` $\gg 3$
Binary number	101010	10101	1010	101
Decimal number	42	21	10	5

Bitwise Operations - Shifts

Shifting a number to the **left** is equivalent to multiplying it with 2^n, where n is the number of shifts.

Shifting a number to the **right** is equivalent to dividing it by 2^n, where n is the number of shifts.

Note: The bits that are shifted out of the number are lost. And a right-shift/division by 2^n is always rounded down to the nearest integer.

Bitwise Operations - Overview

Operator	ASCII	Description
$\wedge$	&	Bitwise AND
$\vee$	|	Bitwise OR
$\oplus$	^	Bitwise XOR
$\neg$	~	Bitwise NOT
$\ll$	<<	Left shift
$\gg$	>>	Right shift

To use bitwise operators in Python, we use the following ASCII-symbols given in the table above.

Bitwise Operations - in python

Usage of binary operators in python:

```
a = 0b0101  # binary representation of 53
b = 0b1011  # binary representation of 43

bin_add = a & b  # 0b0001
bin_or = a | b  # 0b1111
bin_xor = a ^ b  # 0b1110
bin_not = ~a  # 0b1010
bin_shift_left = a << 2  # 0b010100
bin_shift_right = b >> 3  # 0b0001
```

Number systems

Maybe you have noticed the `0b` prefix in the previous example. This is used to indicate that the number is in binary.
Python also supports hexadecimal (`0x`) and octal (`0o`) numbers.
The in-built-functions `oct()`, `bin()`, `int()` and `hex()` can be used to convert between number systems.
`int()` can also be used to convert a string to an integer in a given base.

Number systems - Examples

```
dec = 42
bin_num = bin(dec)  # 0b101010
oct_num = oct(dec)  # 0o52
hex_num = hex(dec)  # 0x2a
# other way around
new_hex = 0xFF
new_oct = 0o77
dec_new_hex = int(new_hex)  # 255
dec_new_oct = int(new_oct)  # 63
# using 17 as a base
base17 = int("F0", base=17)  # 255
```