Съдържание

1 Boolean Values

While the integer, floating-point, and string data types have an unlimited number of possible values, the Boolean data type has only two values: *True* and *False*.

```
>>> spam = True 1
>>> spam
True
>>> true 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined. Did you mean: 'True'?
>>> True = 2 + 2 3
  File "<stdin>", line 1
    True = 2 + 2
    ^^^^
SyntaxError: cannot assign to True
>>>
```

Like any other value, Boolean values are used in expressions and can be stored in variables **(1)**. If you don't use the proper case **(2)** or you try to use True and False for variable names **(3)**, Python will give you an error message.

2 Comparison Operators

Comparison operators compare two values and evaluate down to a single Boolean value.

Table 2-1: Comparison Operators

Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

These operators evaluate to *True* or *False* depending on the values you give them.

```
>>> 42 == 42
True
>>> 42 == 99
False
>>> 2 != 3
True
>>> 2 != 2
False

>>> 'hello' == 'hello'
True
>>> 'hello' == 'Hello'
False
>>> 'dog' != 'cat'
True
>>> True == True
True
>>> True != False
True
>>> 42 == 42.0
True
>>> 42 == '42'
False
```

Note that an integer or floating-point value will always be unequal to a string value. The <, >, <=, *and* >= operators, on the other hand, work properly only with integer and floating-point values.

```
>>> 42 < 100
True
>>> 42 > 100
False
>>> 42 < 42
False
>>> eggCount = 42
❶ >>> eggCount <= 42
True
>>> myAge = 29
❷ >>> myAge >= 10
True
```

3 Boolean Operators

The three Boolean operators (*and*, *or*, and *not*) are used to compare Boolean values. Like comparison operators, they evaluate these expressions down to a Boolean value.

3.1 Binary Boolean Operators

The *and* and *or* operators always take two Boolean values (or expressions), so they're considered *binary* operators. The *and* operator evaluates an expression to *True* if *both* Boolean values are *True*; otherwise, it evaluates to *False*.

```
>>> True and True
True
>>> True and False
False
```

Table 2-2: The and Operator's Truth Table

Expression	Evaluates to...
True and True	True
True and False	False
False and True	False
False and False	False

Table 2-3: The or Operator's Truth Table

Expression	Evaluates to...
True or True	True
True or False	True
False or True	True
False or False	False

Table 2-4: The not Operator's Truth Table

Expression	Evaluates to...
not True	False
not False	True

3.2 Mixing Boolean and Comparison Operators

```
>>> (4 < 5) and (5 < 6)
True
>>> (4 < 5) and (9 < 6)
False
>>> (1 == 2) or (2 == 2)
True
```

The computer will evaluate the left expression first, and then it will evaluate the right expression.

You can also use multiple Boolean operators in an expression, along with the comparison

operators.

```
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
```

Глава 1

If

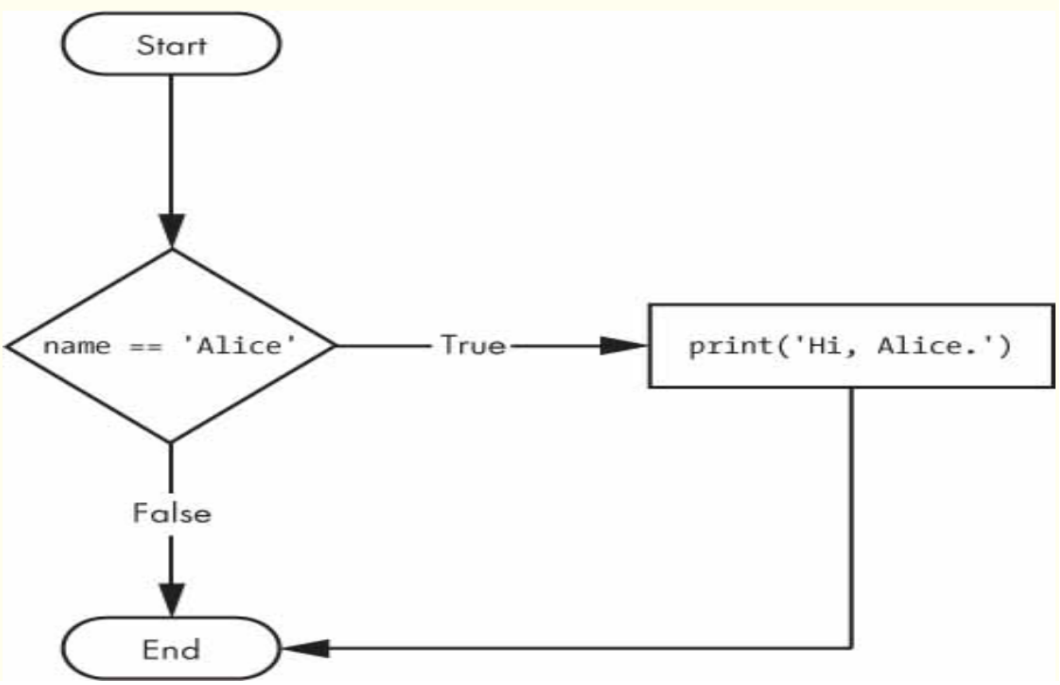
At the heart of every *if statement* is an expression that can be evaluated as *True* or *False* and is called a *conditional test*. Python uses the values *True* and *False* to decide whether the code in an if statement should be executed. If a conditional test evaluates to *True*, Python executes the code following the if statement. If the test evaluates to *False*, Python ignores the code following the if statement.

In plain English, an *if* statement could be read as, “If this condition is true, execute the code in the clause.” In Python, an *if* statement consists of the following:

- The *if* keyword
- A condition (that is, an expression that evaluates to *True* or *False*)
- A colon
- Starting on the next line, an indented block of code (called the *if* clause)

For example:

```
if name == 'Alice':
    print('Hi, Alice.')
```


Start
name == 'Alice'
True
print('Hi, Alice.')
False
End

Глава 2

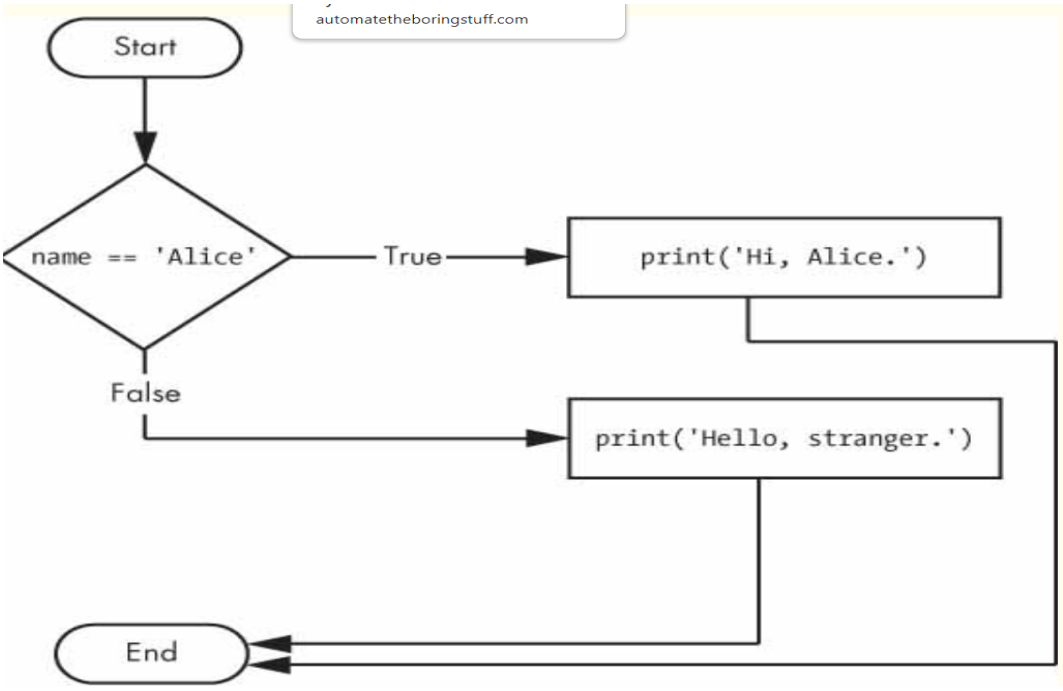
Else

An *if* clause can optionally be followed by an *else* statement. The *else* clause is executed only when the *if* statement's condition is *False*. In plain English, an *else* statement could be read as, "If this condition is true, execute this code. Or else, execute that code." An *else* statement doesn't have a condition, and in code, an *else* statement always consists of the following:

- The *else* keyword
- A colon
- Starting on the next line, an indented block of code (called the *else* clause)

Example:

```
if name == 'Alice':
    print('Hi, Alice.')
else:
    print('Hello, stranger.')
```



1 Живот на променливата

Всяка една променлива си има обхват, в който съществува, наречен **variable scope**. Този обхват уточнява къде една променлива може да бъде използвана. В езика Python променливите могат да бъдат използвани навсякъде, стига да са инициализирани поне веднъж.

В примера по-долу, на последния ред, на който се опитваме да отпечатаме променливата **my_name**, която е дефинирана в **else** конструкцията, ще получим грешка, защото в конкретния случай не се е изпълнило тялото на **else** клаузата, в която инициализираме променливата. Но отпечатването на променливата **can_drive** е безпроблемно, защото програмата е влязва в тялото на **if** клаузата и е инициализирала променливата.

```
my_age = 20
if my_age >= 18:
    can_drive = True
else:
    my_name = 'Ivan'

print(my_age)
print(can_drive)
print(my_name)
```

Глава 3

Elif

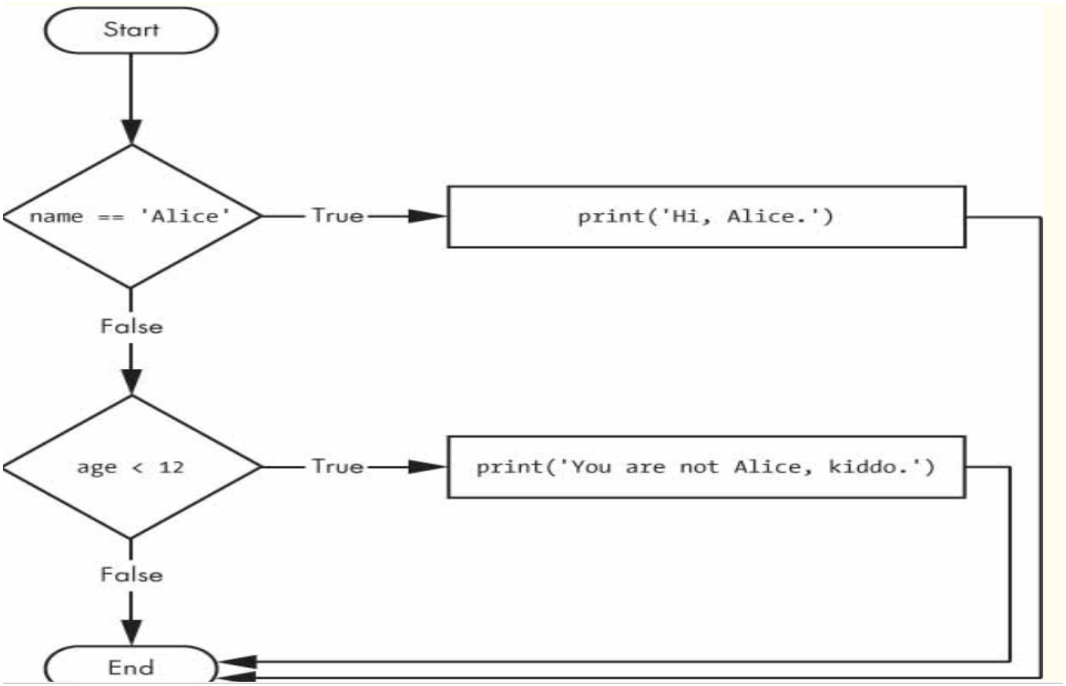
The *elif* statement is an “else if” statement that always follows an *if* or another *elif* statement. It provides another condition that is checked only if all of the previous conditions were *False*. In code, an *elif* statement always consists of the following:

- The *elif* keyword
- A condition (that is, an expression that evaluates to *True* or *False*)
- A colon
- Starting on the next line, an indented block of code (called the *elif* clause)

Example:

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
```

This time, you check the person’s age, and the program will tell them something different if they’re younger than 12.



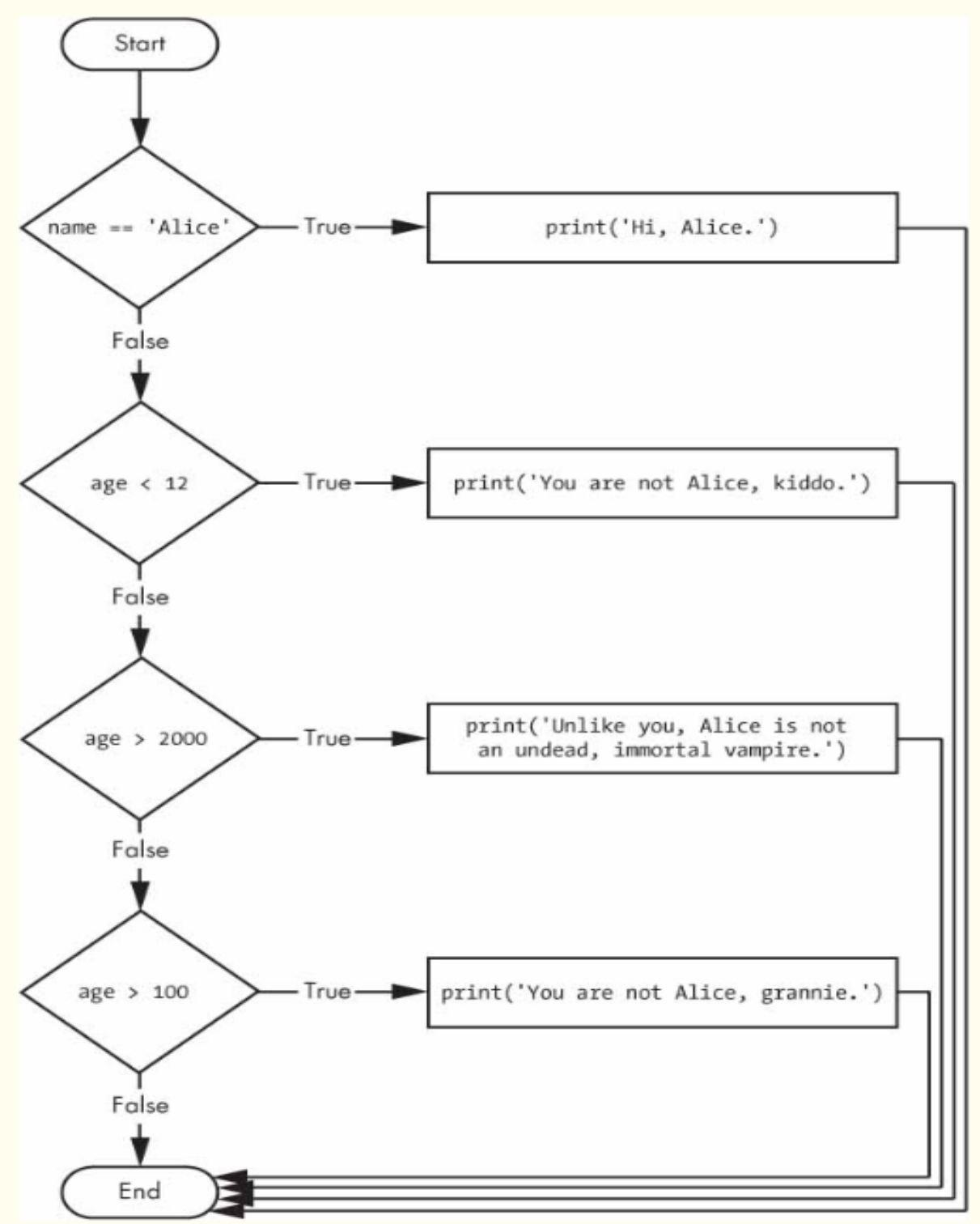
The *elif* clause executes if *age < 12* is *True* and *name == 'Alice' is False*. However, if both of the conditions are *False*, then both of the clauses are skipped. It is not guaranteed that at least one of the clauses will be executed. When there is a chain of *elif* statements, only one or none of the clauses will be executed. Once one of the statements' conditions is found to be *True*, the rest of the *elif* clauses are automatically skipped. For example, open a new file editor window and enter the following code, saving it as *Flow control1.py*:

VSC

```
name = 'Carol'
age = 3000
if name == 'Alice':
    print('Hi, Alice.')
elif age > 12:
    print('You are not Alice, kiddo.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')
elif age > 100:
    print('You are not Alice, grannie.')
```

Result:

```
Unlike you, Alice is not an undead, immortal vampire.
```



The order of the *elif* statements does matter, however. Let's rearrange them to introduce a bug. Remember that the rest of the *elif* clauses are automatically skipped once a *True* condition has been found, so if you swap around some of the clauses in *Flow control1.py*, you run into a problem. Change the code to look like the following, and save it as *Flow control2.py*:

VSC

```
name = 'Carol'
age = 3000
if name == 'Alice':
```

```
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
elif age > 100:
    print('You are not Alice, grannie.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')
```

You might expect the code to print the string *'Unlike you, Alice is not an undead, immortal vampire.'*. However, because the *age > 100* condition is *True* (after all, 3,000 is greater than 100), the string *'You are not Alice, grannie.'* is printed, and the rest of the *elif* statements are automatically skipped.

Optionally, you can have an *else* statement after the last *elif* statement. In that case, it is guaranteed that at least one (and only one) of the clauses will be executed. If the conditions in every *if* and elif statement are *False*, then the *else* clause is executed.

Example:

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
else:
    print('You are neither Alice nor a little kid.')
```

```
name = ''
while name != 'your name':
    print('Please type your name.')
    name = input()
print('Thank you!')
```

Глава 4

While

With the *while* loop we can execute a set of statements as long as a condition is true.

```
>>> i = 1
>>> while i < 6:
...     print(i)
...     i += 1
...
1
2
3
4
5
>>>
```

Print i as long as i is less than 6.

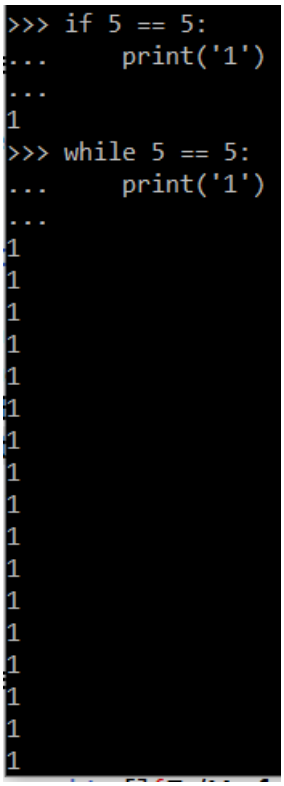
A *while* statement always consists of the following:

- The *while* keyword
- A condition (that is, an expression that evaluates to *True* or *False*)
- A colon
- Starting on the next line, an indented block of code (called the *while* clause)

At the end of an *if* clause, the program execution continues after the *if* statement. But at the end of a *while* clause, the program execution jumps back to the start of the *while* statement.

Note: remember to increment[1] *i*, or else the loop will continue forever:

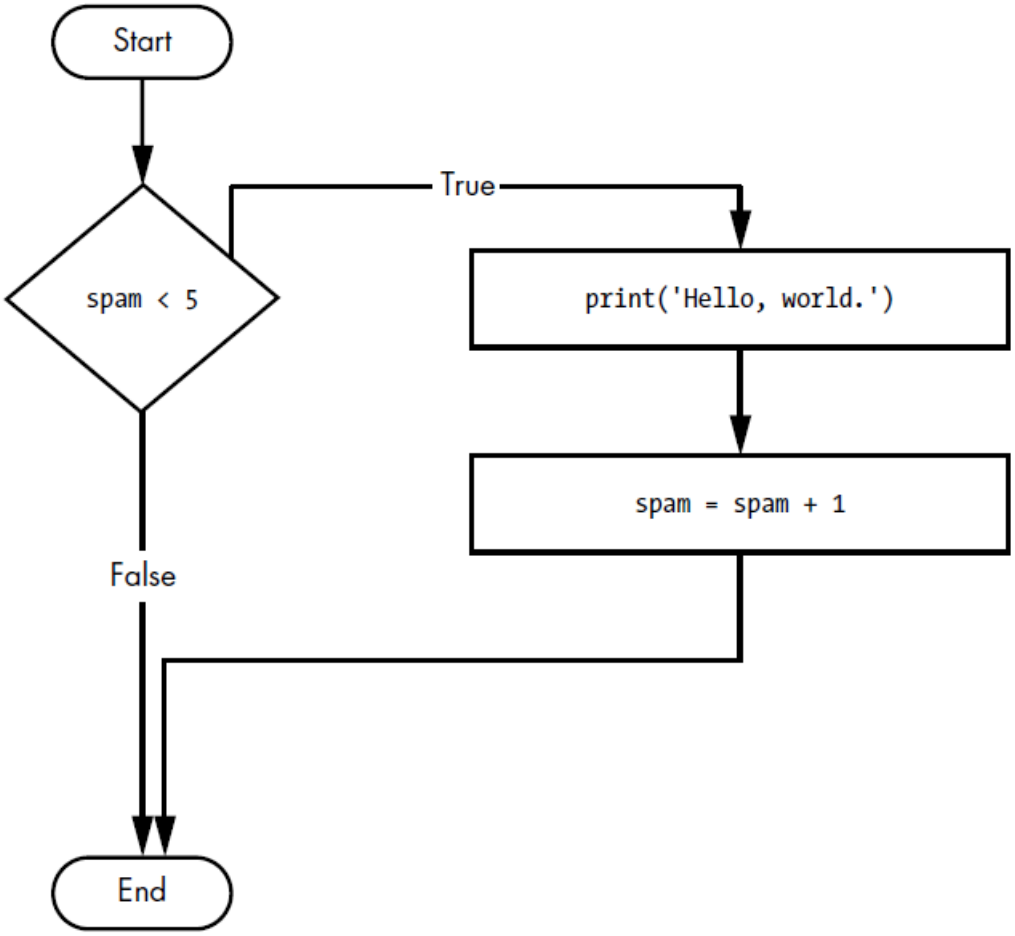
[1] an increase or addition, especially one of a series on a fixed scale.



...

Example of *while* statement and its flow chart.

```
spam = 0
while spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

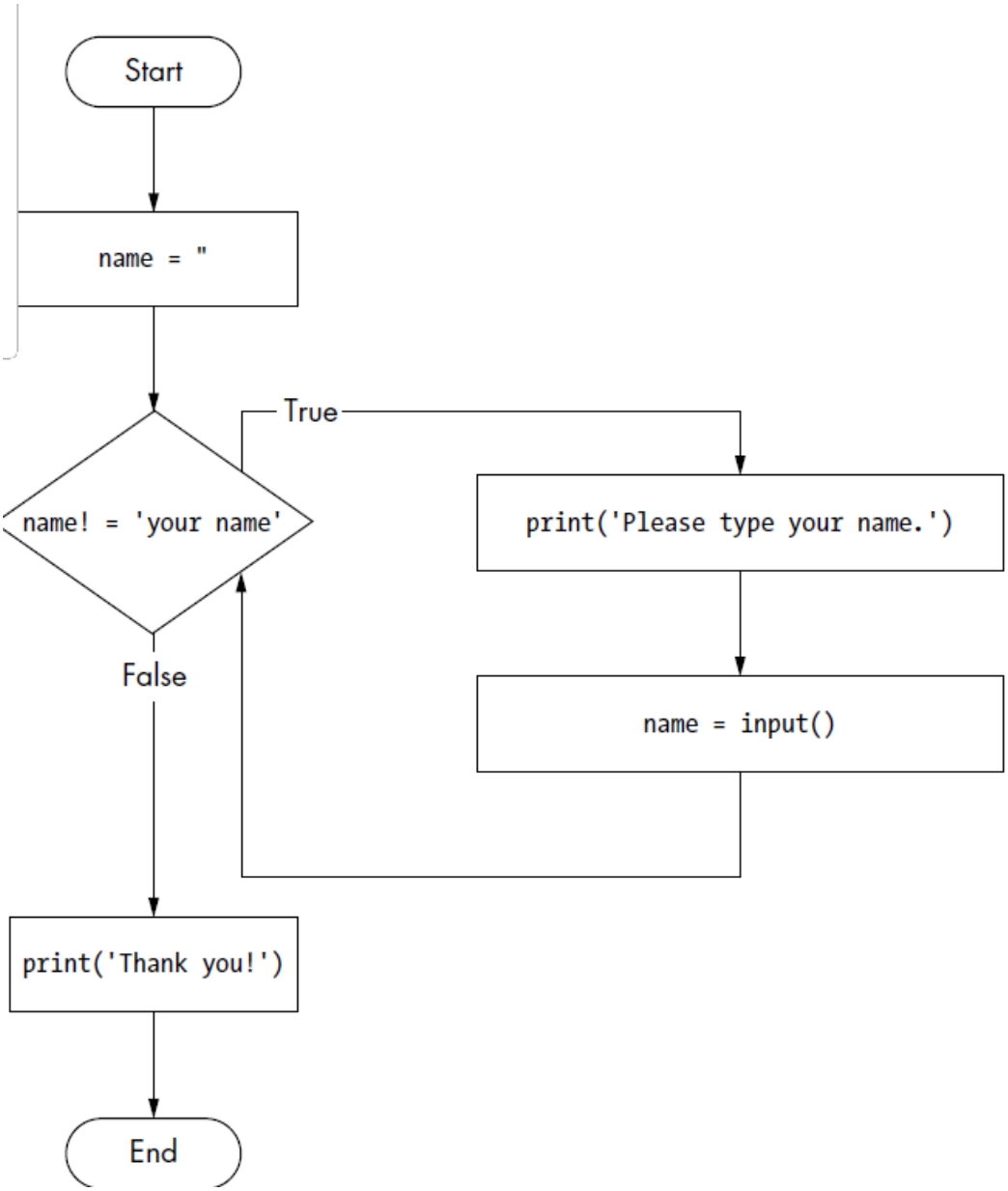


The code with the *while loop*, on the other hand, will print five times. It stops after five prints because the integer in *spam* is incremented by one at the end of each loop iteration, which means that the loop will execute five times before *spam* < *5* is *False*.

In the *while* loop, the condition is always checked at the start of each iteration (that is, each time the loop is executed). If the condition is *True*, then the clause is executed, and afterward, the condition is checked again. The first time the condition is found to be *False*, the while clause is skipped.

Another example:

```
name = ''
while name != 'your name':
    print('Please type your name.')
    name = input()
print('Thank you!')
```



[2]

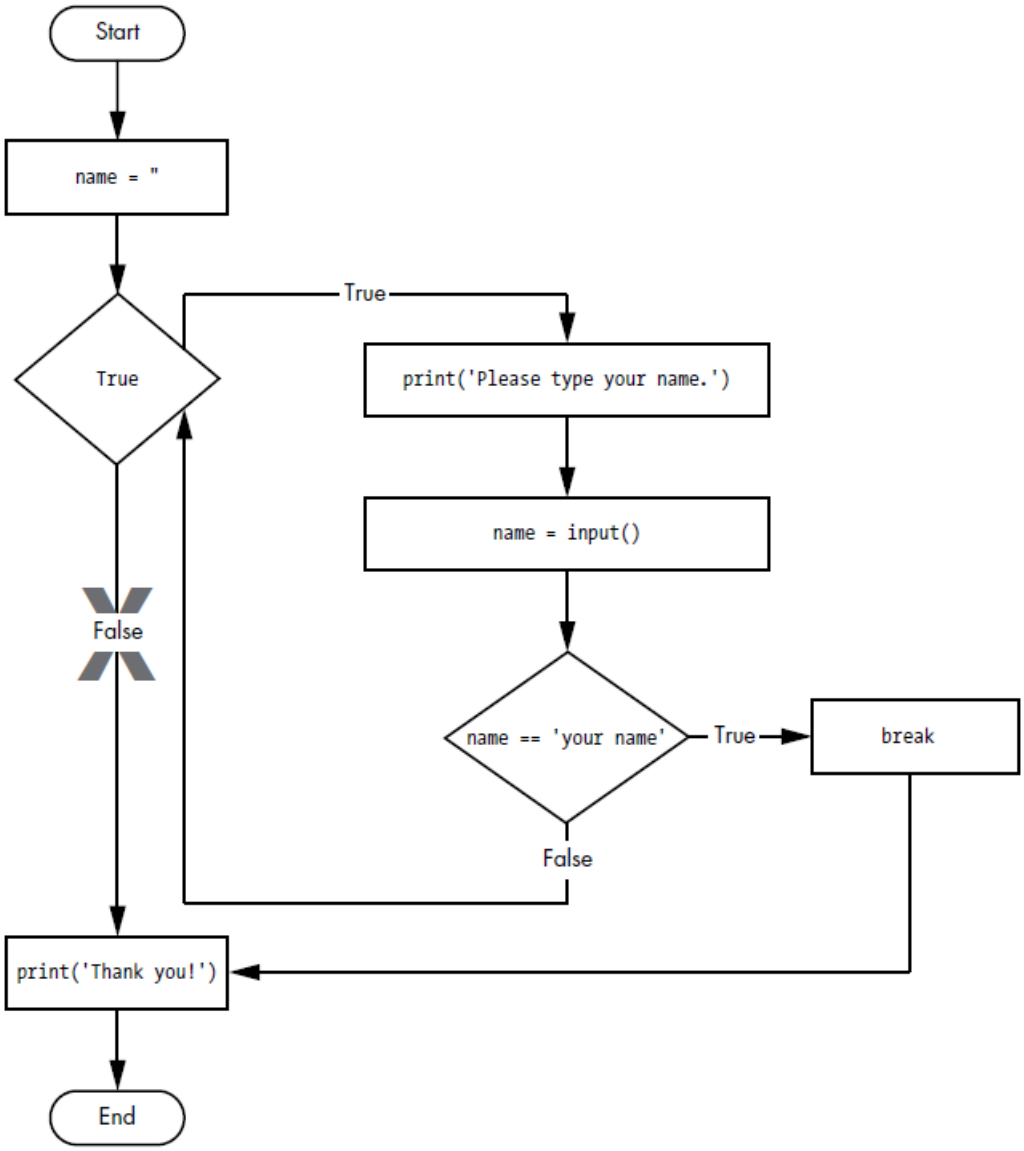
1 Break

There is a shortcut to getting the program execution to break out of a while loop's clause early. Here's a program that does the same thing as the previous program, but it uses a *break* statement to escape the loop.

```
❶ while True:
      print('Please type your name.')
❷     name = input()
❸     if name == 'your name':
❹         break
❺ print('Thank you!')
```

[2] Automate the Boring Stuff with Python, Al Sweigart

The first line **(1)** creates an *infinite loop*; it is a *while* loop whose condition is always *True*. Just like before, this program asks the user to type your *name* **(2)**. Now, however, while the execution is still inside the *while* loop, an if statement gets executed **(3)** to check whether *name* is equal to your name. If this condition is *True*, the *break* statement is run **(4)**, and the execution moves out of the loop to *print('Thank you!')* **(5)**. Otherwise, the *if* statement's clause with the *break* statement is skipped, which puts the execution at the end of the *while* loop. At this point, the program execution jumps back to the start of the while statement **(1)** to recheck the condition. Since this condition is merely the *True* Boolean value, the execution enters the loop to ask the user to type your *name* again.

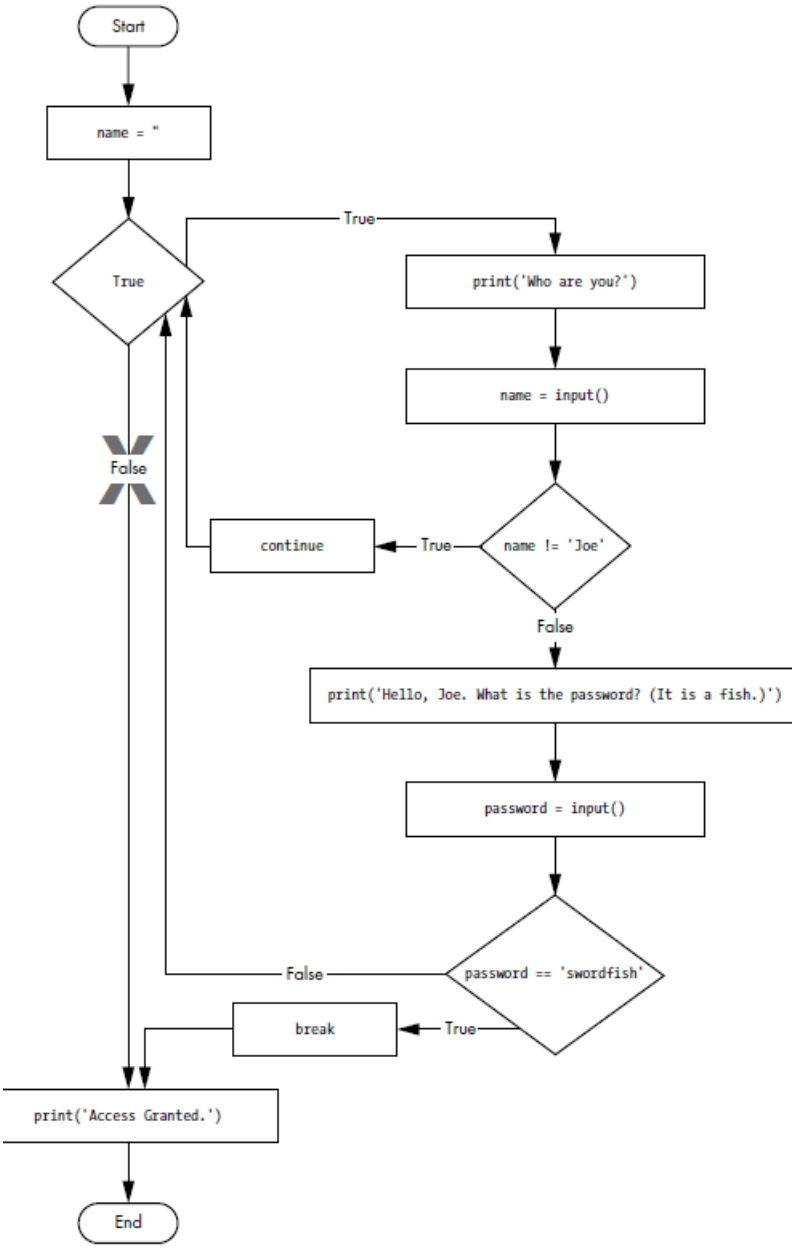


2 Continue

When the program execution reaches a *continue* statement, the program execution immediately jumps back to the start of the loop and reevaluates the loop's condition.

```
while True:
    print('Who are you?')
    name = input()
❶   if name != 'Joe':
❷       continue
    print('Hello, Joe. What is the password? (It is a fish.)')
❸   password = input()
    if password == 'swordfish':
❹       break
❺ print('Access granted.')
```

If the user enters any name besides *Joe* **(1)**, the *continue* statement **(2)** causes the program execution to jump back to the start of the loop. When it reevaluates the condition, the execution will always enter the loop, since the condition is simply the value *True*. Once they make it past that if statement, the user is asked for a password **(3)**. If the password entered is *swordfish*, then the *break* statement **(4)** is run, and the execution jumps out of the while loop to print *Access granted* **(5)**.



Глава 5

For

A *for* loop is used for iterating over a sequence (that is either a *list, a tuple, a dictionary, a set, or a string*).

With the *for* loop we can execute a set of statements, once for each item in a list, tuple, set etc.

```
>>> fruits = ['apple', 'banana', 'cherry']
>>> for x in fruits:
...     print(x)
...
apple
banana
cherry
>>>
```

Print each fruit in a fruit list.

The *for* loop does not require an indexing variable to set beforehand[1]

Нека разгледаме един пример за **for** цикъл, който преминава последователно през числата от 1 до 10 и ги отпечатва:

```
>>> for i in range(1, 11):
...     print('i = ' + str(i))
...
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
>>>
```

Цикълът започва с оператора **for** и преминава през всички стойности за дадена

[1] (w3school)

променлива в даден интервал, например всички числа от 1 до 10 включително (без да включва 11), и за всяка стойност изпълнява поредица от команди.

В декларацията на цикъла може да се зададе начална стойност и крайна стойност, като крайната стойност не е включена в диапазона. Тялото на цикъла пред-ставлява блок с една или няколко команди.

В повечето случаи един **for** цикъл се завърта от **1** до **n** (например от 1 до 10). Целта на цикъла е да се премине последователно през числата 1, 2, 3, . . . , n и за всяко от тях да се изпълни някакво действие. В примера по-горе променливата **i** приема стойности от 1 до 10 и в тялото на цикъла се отпечатва текущата стойност. Цикълът се повтаря 10 пъти и всяко от тези повторения се нарича "*итерация*".

1 range()

range(start, stop, step)

Create a sequence of numbers from 3 to 5, and print each item in the sequence:

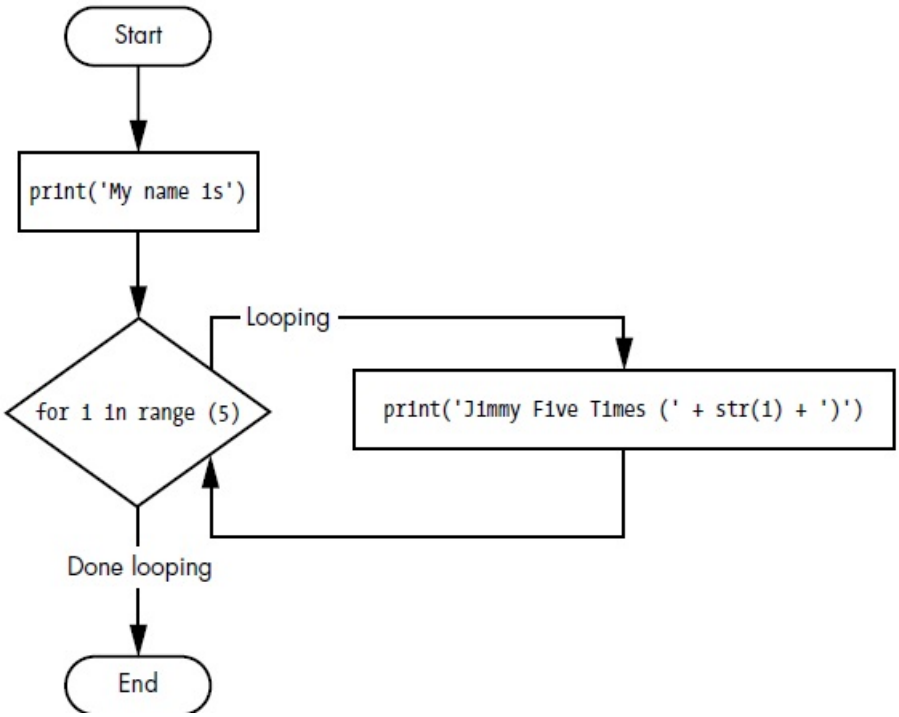
```
>>> x = range(3, 6)
>>> for n in x:
...   print(n)
...
3
4
5
```

Create a sequence of numbers from 3 to 19, but increment by 2 instead of 1:

```
>>> x = range(3, 20, 2
>>> for n in x:
...     print(n)
...
3
5
7
9
11
13
15
17
19
```

```
print('My name is')
for i in range(5):
    print('Jimmy Five Times (' + str(i) + ')')
```

The code in the `for` loop's clause is run five times. The first time it is run, the variable `i` is set to `0`. The `print()` call in the clause will print `Jimmy Five Times (0)`. After Python finishes an iteration through all the code inside the `for` loop's clause, the execution goes back to the top of the loop, and the `for` statement increments `i` by one. This is why `range(5)` results in five iterations through the clause, with `i` being set to `0`, then `1`, then `2`, then `3`, and then `4`. The variable `i` will go up to, but will not include, the integer passed to `range()`. Figure 2-14 shows a flowchart for the *fiveTimes.py* program.



You can use break and continue statements inside for loops as well. The continue statement will continue to the next value of the for loop's counter, as if the program execution had reached the end of the loop and returned to the start. In fact, you can use continue and break statements only inside while and for loops. If you try to use these statements elsewhere, Python will give you an error.