Control Structures in Python

Anab Batool Kazmi

control structures

- These are compound statements that ***alter program control flow***
- Control flow refers to the sequence, a program will follow during its execution.
- Python has three types of control structures
  - **Sequential** - default mode
  - **Selection** - used for decisions and branching
  - **Repetition** - used for looping, i.e., repeating a piece of code multiple times.

Sequential statements

- Sequential statements are a set of statements whose execution process happens in a sequence.
- The problem with sequential statements is that if the logic has broken in any one of the lines, then the complete source code execution will break.

```
1  ## This is a Sequential statement
2
3  a=20
4  b=10
5  c=a-b
6  print("Subtraction is : ",c)
```

Selection/Decision control statements

Selection/Decision control statements

- In Python, the selection statements are also known as **Decision control statements** or **branching statements**.
- The selection statement **allows a program to test several conditions** and **execute instructions based on which condition is true**.
- Some decision control statements are:
  - ***if***
  - ***if-else***
  - ***nested if***
  - ***if-elif-else***

Python

```
if <expr>:
    <statement>
```

if statement

- **<expr>** is an expression evaluated in a **Boolean context**
- **<statement>** is a valid Python statement, which must be indented.
- If **<expr> is true** (evaluates to a value that is "**truthy**"), then <statement> is executed. If <expr> is false, then <statement> is skipped over and not executed.

- Note that the **colon (:)** following <expr> is required.

if statement

```
>>> x = 0
>>> y = 5

>>> if x < y:                                  # Truthy
...     print('yes')
...
yes
>>> if y < x:                                  # Falsy
...     print('yes')
...

>>> if x:                                      # Falsy
...     print('yes')
```

```
>>> if y:                                      # Truthy
...     print('yes')
...
yes

>>> if x or y:                                 # Truthy
...     print('yes')
...
yes
>>> if x and y:                                # Falsy
...     print('yes')
...

>>> if 'aul' in 'grault':                      # Truthy
...     print('yes')
...
yes
>>> if 'quux' in ['foo', 'bar', 'baz']:  # Falsy
...     print('yes')
...
```

if-else statement

- you want to evaluate a condition and take one path if it is true but specify an alternative path if it is not.

Python

```
if <expr>:
    <statement(s)>
else:
    <statement(s)>
```

if-else statement

- Portions of a conditional expression are not evaluated if they don’t need to be.

- In the expression <expr1> if <conditional_expr> else <expr2>:

- If <conditional_expr> is true, <expr1> is returned and <expr2> is not evaluated.
- If <conditional_expr> is false, <expr2> is returned and <expr1> is not evaluated.

if-else statement

Python

```
>>> x = 20

>>> if x < 50:
...     print('(first suite)')
...     print('x is small')
... else:
...     print('(second suite)')
...     print('x is large')
...
(first suite)
x is small
```

Python

```
>>> x = 120
>>>
>>> if x < 50:
...     print('(first suite)')
...     print('x is small')
... else:
...     print('(second suite)')
...     print('x is large')
...
(second suite)
x is large
```

Nested if

- Nested if statements are an if statement inside another if statement.

```
1  a = 20
2  b = 10
3  c = 15
4  if a > b:
5      if a > c:
6          print("a value is big")
7      else:
8          print("c value is big")
```

if-elif-else

- used for branching execution based on **several alternatives.**
- Python evaluates each <expr> in turn and executes the suite corresponding to the first that is true.
- If none of the expressions are true, and an else clause is specified, then its suite is executed
- An arbitrary number of elif clauses can be specified. The else clause is optional. If it is present, there can be only one, and it must be specified last
- At most, one of the code blocks specified will be executed. If an else clause isn't included, and all the conditions are false, then none of the blocks will be executed.

Python

```
if <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
    ...
else:
    <statement(s)>
```

if-elif-else

Python

```
>>> name = 'Joe'
>>> if name == 'Fred':
...     print('Hello Fred')
... elif name == 'Xander':
...     print('Hello Xander')
... elif name == 'Joe':
...     print('Hello Joe')
... elif name == 'Arnold':
...     print('Hello Arnold')
... else:
...     print("I don't know who you are!")
...
Hello Joe
```

One-Line if Statements

- it is permissible to write an entire if statement on one line.

```
Python

if <expr>: <statement>
```

- There can even be more than one <statement> on the same line, separated by semicolons:

```
Python

if <expr>: <statement_1>; <statement_2>; ...; <statement_n>
```

- If <expr> is true, execute all of <statement_1> … <statement_n>. Otherwise, don’t execute any of them.

One-Line if Statements

Python

```
>>> if 'f' in 'foo': print('1'); print('2'); print('3')
...
1
2
3
>>> if 'z' in 'foo': print('1'); print('2'); print('3')
...
```

One-Line if Statements

- Multiple statements may be specified on the same line as an elif or else clause as well:

Python

```
>>> x = 2
>>> if x == 1: print('foo'); print('bar'); print('baz')
... elif x == 2: print('qux'); print('quux')
... else: print('corge'); print('grault')
...
qux
quux

>>> x = 3
>>> if x == 1: print('foo'); print('bar'); print('baz')
... elif x == 2: print('qux'); print('quux')
... else: print('corge'); print('grault')
...
corge
grault
```

Conditional Expressions (Python's Ternary Operator)

- This is different from the if statement forms listed previously because **it is not a control structure that directs the flow of program execution**.
- It **acts more like an operator** that defines an expression.

```
Python

<expr1> if <conditional_expr> else <expr2>
```

- In the above example, **<conditional_expr> is evaluated first**. If it is true, the expression evaluates to <expr1>. If it is false, the expression evaluates to <expr2>.

Conditional Expressions (Python's Ternary Operator)

Python

```
>>> raining = False
>>> print("Let's go to the", 'beach' if not raining else 'library')
Let's go to the beach
>>> raining = True
>>> print("Let's go to the", 'beach' if not raining else 'library')
Let's go to the library

>>> age = 12
>>> s = 'minor' if age < 21 else 'adult'
>>> s
'minor'

>>> 'yes' if ('qux' in ['foo', 'bar', 'baz']) else 'no'
'no'
```

Conditional Expressions (Python's Ternary Operator)

- A common use of the conditional expression is to select variable assignment.
- The conditional expression has lower precedence than virtually all the other operators

Conditional Expressions (Python's Ternary Operator)

- the + operator binds more tightly than the conditional expression, so 1 + x and y + 2 are evaluated first, followed by the conditional expression.
- The parentheses in the second case are unnecessary and do not change the result:

Python

```
>>> x = y = 40

>>> z = 1 + x if x > y else y + 2
>>> z
42

>>> z = (1 + x) if x > y else (y + 2)
>>> z
42
```

Conditional Expressions (Python's Ternary Operator)

- If you want the conditional expression to be evaluated first, you need to surround it with grouping parentheses.
- (x if x > y else y) is evaluated first.
- The result is y, which is 40, so z is assigned 1 + 40 + 2 = 43

Python

```
>>> x = y = 40

>>> z = 1 + (x if x > y else y) + 2
>>> z
43
```

The Python pass Statement

- Occasionally, you may find that you want to write what is called a **code stub**: **a placeholder for where you will eventually put a block of code that you haven't implemented yet.**

- In languages where token delimiters are used to define blocks, like the curly braces in Perl and C, empty delimiters can be used to define a code stub. For example, the following is legitimate Perl or C code:

Perl

```
# This is not Python
if (x)
{
}
```

The Python pass Statement

```python
if True:
    pass

print('foo')
```

- Because Python uses indentation instead of delimiters, it is not possible to specify an empty block.
- If you introduce an if statement with if <expr>:, something has to come after it, either on the same line or indented on the following line.
- The Python pass statement solves this problem. It doesn't change program behavior at all. It is used as a placeholder to keep the interpreter happy in any situation where a statement is syntactically required, but you don't really want to do anything

Repetition /Iteration control structure

Iteration

- Iteration means executing the same block of code over and over, potentially many times. A programming structure that implements iteration is called a **loop**.

- In programming, there are two types of iteration, indefinite and definite:

  - With **indefinite iteration**, the number of times the loop is executed isn't specified explicitly in advance. Rather, the designated block is executed repeatedly as long as some condition is met.

  - With **definite iteration**, the number of times the designated block will be executed is specified explicitly at the time the loop starts.

indefinite iteration - The while Loop

- When a while loop is encountered, <expr> is first evaluated in Boolean context. If it is true, the loop body is executed.
- Then <expr> is checked again, and if still true, the body is executed again.
- This continues until <expr> becomes false, at which point program execution proceeds to the first statement beyond the loop body.

Python

```
while <expr>:
    <statement(s)>
```

indefinite iteration - The while Loop

Python

```
>>> n = 5
>>> while n > 0:
...     n -= 1
...     print(n)
...
```

Python

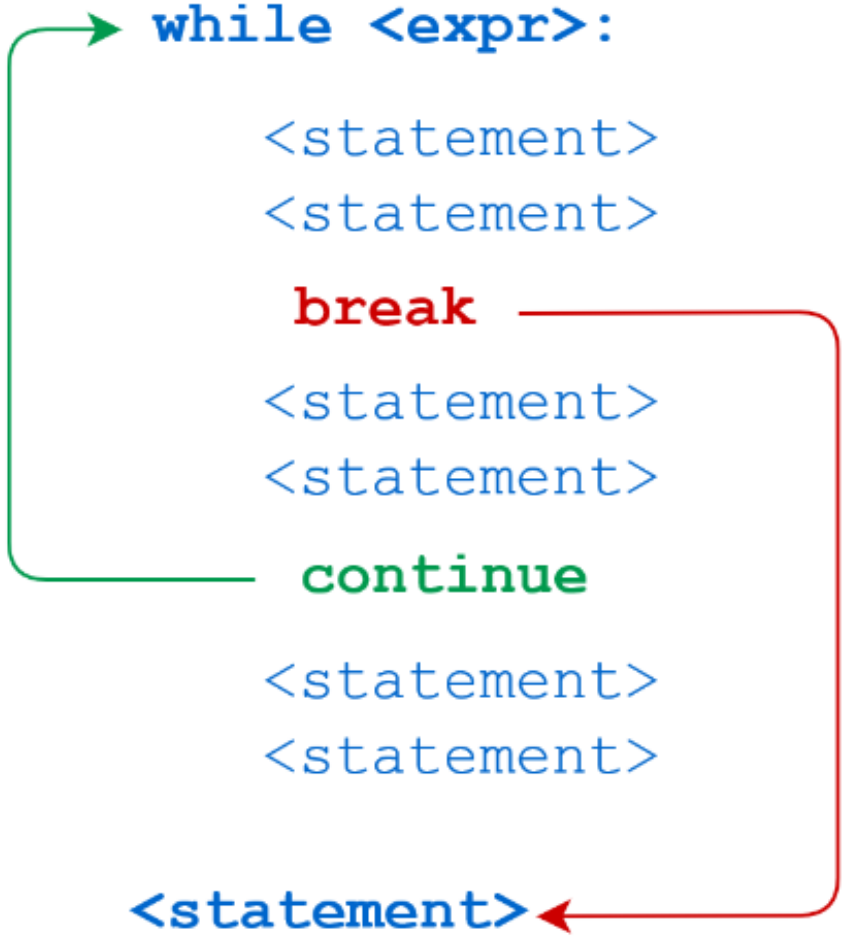
```
>>> n = 0
>>> while n > 0:
...     n -= 1
...     print(n)
```

The Python break and continue Statements

- Python provides two keywords that terminate a loop iteration prematurely:

- The Python **break statement immediately terminates a loop entirely.** Program execution proceeds to the first statement following the loop body.

- The Python **continue statement immediately terminates the current loop iteration**. Execution jumps to the top of the loop, and the controlling expression is re-evaluated to determine whether the loop will execute again or terminate.



break and continue

break statement

Python

```python
n = 5
while n > 0:
    n -= 1
    if n == 2:
        break
    print(n)
print('Loop ended.')
```

Windows Command Prompt

```
C:\Users\john\Documents>python break.py
4
3
Loop ended.
```

continue statement

Python

```python
n = 5
while n > 0:
    n -= 1
    if n == 2:
        continue
    print(n)
print('Loop ended.')
```

Windows Command Prompt

```
C:\Users\john\Documents>python continue.py
4
3
1
0
Loop ended.
```

The else Clause

- Python allows an optional else clause at the end of a while loop. This is a unique feature of Python, not found in most other programming languages.
- The <additional_statement(s)> specified in the else clause will be executed when the while loop terminates.

Python

```
while <expr>:
    <statement(s)>
<additional_statement(s)>
```

Python

```
while <expr>:
    <statement(s)>
else:
    <additional_statement(s)>
```

The else Clause

- In this case, without the else clause, <additional_statement(s)> will be executed after the while loop terminates, no matter what.

Python

```
while <expr>:
    <statement(s)>
<additional_statement(s)>
```

The else Clause

- When <additional_statement(s)> are placed in an else clause, they will be executed only if the loop terminates “by exhaustion”—that is, if the loop iterates until the controlling condition becomes false. If the loop is exited by a break statement, the else clause won’t be executed.

Python

```
while <expr>:
    <statement(s)>
else:
    <additional_statement(s)>
```

The else Clause

Python

```
>>> n = 5
>>> while n > 0:
...     n -= 1
...     print(n)
... else:
...     print('Loop done.')
...
4
3
2
1
0
Loop done.
```

Python

```
>>> n = 5
>>> while n > 0:
...     n -= 1
...     print(n)
...     if n == 2:
...         break
... else:
...     print('Loop done.')
...
4
3
2
```

One-Line while Loops

- As with an if statement, a while loop can be specified on one line. If there are multiple statements in the block that makes up the loop body, they can be separated by semicolons (;):

Python

```
>>> n = 5
>>> while n > 0: n -= 1; print(n)

4
3
2
1
0
```

One-Line while Loops

- This only works with simple statements though. You can’t combine two compound statements into one line. Thus, you can specify a while loop all on one line as above, and you write an if statement on one line:

Python

```
>>> while n > 0: n -= 1; if True: print('foo')
SyntaxError: invalid syntax
```

Definite iteration loops in Python

- **Definite iteration loops** are frequently referred to as **for loops** because **for is the keyword** that is used to introduce them in nearly all programming languages, including Python.
- Python for loop is the collection-Based or Iterator-Based Loop
- This type of loop iterates over a collection of objects, rather than specifying numeric values or conditions
- Each time through the loop, **the variable var takes on the value of the next object in <iterable>.**

```
Python

for <var> in <iterable>:
    <statement(s)>
```

For loop

- <iterable> is a collection of objects—for example, a list or tuple.
- The <statement(s)> in the loop body are denoted by indentation, as with all Python control structures, and are executed once for each item in <iterable>.
- The loop variable <var> takes on the value of the next element in <iterable> each time through the loop.

Python

```
for <var> in <iterable>:
    <statement(s)>
```

For loop

```Python
>>> a = ['foo', 'bar', 'baz']
>>> for i in a:
...     print(i)
...
foo
bar
baz
```

- In this example, **<iterable> is the list a, and <var> is the variable i.**
- Each time through the loop, i takes on a successive item in a, so print() displays the values 'foo', 'bar', and 'baz', respectively.

The range() Function

- the built-in range() function, which returns an iterable that yields a sequence of integers.
- range(<end>) returns an iterable that yields integers starting with 0, up to but not including <end>:

```
>>> x = range(5)
>>> x
range(0, 5)
>>> type(x)
<class 'range'>
```

Python

```
>>> for n in x:
...     print(n)
...
0
1
2
3
4
```

The range() Function

- The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

```
range(start, stop, step)
```

Parameter Values

Parameter	Description
start	Optional. An integer number specifying at which position to start. Default is 0
stop	Required. An integer number specifying at which position to stop (not included).
step	Optional. An integer number specifying the incrementation. Default is 1

The range() Function

```
x = range(3, 20, 2)

for n in x:
  print(n)
```

```
3
5
7
9
11
13
15
17
19
```