

00001101

CHAPTER 13

Control Flow

Ch-13: Control Flow

Learning Objectives

- *Define the term control flow*
- *State the purpose of control flow statements*
- *State the purpose of conditional expressions*
- *State the purpose of the if statement*
- *Describe scenarios when you would use an if statement*
- *State the purpose of the if statement's elif and else clauses*
- *Demonstrate your ability to write nested if statements using if/elif/else*
- *State the purpose of the for statement*
- *Explain the purpose of the range() function*
- *Explain what is meant by the term iterable*
- *State the purpose of the while statement*
- *Explain why you would use an else clause in a for or while loop*
- *State the purpose of the match statement*
- *State the purpose of the break statement*

0
0
0
0
1
1
0
1

INTRODUCTION

To do anything remotely useful in life you must make decisions based on available data, execute tasks repeatedly either forever or for a set amount of time, know when to stop doing something and when to keep going, and know how to judge whether or not something someone says to you is true or false. The same concepts apply to programming. The constructs provided by Python that enable you to write this type of code are referred to as *control flow* statements.

Python provides several types of control flow statements that enable you to write useful programs. These include *branching* and *looping* statements. For branching, Python provides the `if` statement along with its `elif` and `else` clauses. The `match` statement is also used for branching as well as pattern matching. For looping, Python provides the `while` and `for` statements.

The common thread shared among branching and looping statements is the notion of a *condition*. Branching statements determine which path to take based on the result of a *conditional expression*. Looping statements, too, use the results of conditional expressions to determine when to exit the loop. To effectively employ branching and looping statements in your programs you must master the formulation of *conditional expressions*.

Along the way you'll learn new vocabulary and concepts including what is meant by the term *iterator* and *iterating*. If you have experience in programming languages like C++, Java, or C#, you may need to break some old habits regarding how to write for loops.

When you have finished this chapter, you'll be able to write Python programs that process data, make decisions, and produce meaningful results.

1 CONDITIONAL EXPRESSIONS

Conditional expressions are a fundamental concept central to the mastery of control flow statements. Essentially, all control flow statements must evaluate a conditional expression to determine which code path to follow. In Python, the conditional expression may be explicitly stated or implied.

1.1 WHAT IS TRUE? WHAT IS FALSE?

To understand conditional expressions you must first learn what sorts of things are considered true and what sorts of things are considered false. Many types of Python objects can be individually tested to determine if they are true or false; objects that evaluate to true are referred to as being *truthy* and those that evaluate to false are considered *falsy*. Some of the differences between what Python considers truthy and falsy are obvious, as is the case with the Python boolean literals `True` and `False`. `True` is always true and `False` is always false. Likewise, `1` is true and `0` is false. Just knowing this little bit gets you pretty far. The rest can be learned over time and if you don't try anything too fancy in your code without first doing some research you should be fine.

Generally speaking, when you write code, you will find yourself performing two types of object comparisons: 1. comparing objects against each other, and 2. using individual objects in conditional expressions. Example 13-1 gives the code for a short program that shows the results of common Python object comparisons and shows whether or not they are truthy or falsy.

```

1  from prettytable import PrettyTable
2
3  def main():
4      x = PrettyTable()
5      x.field_names = ["Object", "Truthy/Falsy"]
6
7      x.add_row(["True == True", (True == True)])
8      x.add_row(["False == True", (False == True)])
9      x.add_row(["1 == True", (1 == True)])
10     x.add_row(["0 == True", (0 == True)])
11     x.add_row(["-1 == True", (-1 == True)])
12     x.add_row(["2 == True", (2 == True)])
13     x.add_row(["[] == True", ([] == True)])
14     x.add_row(["[][] == True", ([[ ]] == True)])
15     print(x)
16
17     print()
18
19     y = PrettyTable()
20     y.field_names = ["Object", "Truthy/Falsy"]
21     y.add_row(["if True:", True if True else False])
22     y.add_row(["if 1:", True if 1 else False])
23     y.add_row(["if 2:", True if 2 else False])
24     y.add_row(["if -1:", True if -1 else False])
25     y.add_row(["if 0:", True if 0 else False])
26     y.add_row(["if []:", True if [] else False])
27     y.add_row(["if [[]]:", True if [[]] else False])
28     y.add_row(["if {}: ", True if {} else False])
29     y.add_row(["if {'key': 'value'}:", True if {'key': 'value'} else False])
30     y.add_row(["if '':", True if '' else False])
31     y.add_row(["if 'hello':", True if 'hello' else False])
32     print(y)
33
34
35     if __name__ == "__main__":
36         main()
37

```

Referring to example 13.1 — On line 1, I am importing a library called `prettytable` that makes it easy to create tabular data output. Overall, this program consists of the `main()` function which starts on line 3. Lines 7 through 14 compare various objects to the boolean literal `True`. The `==` operator performs an equality test. Lines 20 through 31 use various objects in a conditional expression. When you run this program and study the output you can make some general assumptions about what Python considers `truthy` or `falsy`. Figure 13-1 shows the results of running this program.

Referring to figure 13-1 — The first table shows that when comparing objects directly against the boolean literal `True`, only itself and the numeric value `1` evaluate to `True`. `False` and the numeric value `0` evaluate to `False`, as do all the other objects. The bottom table shows that when used in an `if` statement, `True` evaluates to `True`, as expected, as does the numeric value `1`, as expected, but so do the values `2` and `negative 1`. `False`, and the numeric value `0` evaluates to `False`, also as expected. An empty list `[]` evaluates to `False` but a non-empty list `[[]]` evaluates to `True`. An empty dictionary `{ }` evaluates to `False` but a non-empty dictionary `{ 'key': 'value' }` evaluates to `True`. An empty string `''` evaluates to `False` but a non-empty string `'hello'` evaluates to `True`.

```

[516:16] swodog@macos-mojave-testbed $ clear

Sun Jan  7 07:06:39 EST 2024
~/dev/cst_with_python_1st_ed/chapter13/TruthyFalsy (main)
[517:17] swodog@macos-mojave-testbed $ python3 truthyfalse.py

+-----+
| Object | Truthy/Falsy |
+-----+
| True == True | True |
| False == True | False |
| 1 == True | True |
| 0 == True | False |
| -1 == True | False |
| 2 == True | False |
| [] == True | False |
| [] == True | False |
+-----+

+-----+
| Object | Truthy/Falsy |
+-----+
| if True: | True |
| if 1: | True |
| if 2: | True |
| if -1: | True |
| if 0: | False |
| if []: | False |
| if []: | True |
| if {}: | False |
| if {'key': 'value'}: | True |
| if '': | False |
| if 'hello': | True |
+-----+

```

Figure 13-1: Results of Running Example 13-1

Pro Tip: Be aware of what Python considers truthy and falsy when formulating conditional expressions. Always verify the truthy or falsy behavior of an object before relying on its behavior in your code.

1.2 COMPARISON OPERATORS

Table 13-1 lists Python’s comparison operators. Use comparison operators when you need to compare the values of different objects. You’ve already seen the equality operator `==` in action in many examples up to this point in the book.

Operator	Name	Example
<code>==</code>	Equals	<code>count == 23</code>
<code>!=</code>	Not Equals	<code>name != "Steve"</code>
<code>></code>	Greater Than	<code>red_states > blue_states</code>
<code><</code>	Less Than	<code>account["value"] < minimum_balance</code>
<code>>=</code>	Greater Than or Equal To	<code>len(string_var) >= 10</code>
<code><=</code>	Less Than or Equal To	<code>days_remaining <= 1</code>

Table 13-1: Python Comparison Operators

1.3 LOGICAL OPERATORS

Table 13-2 lists Python's logical operators. Use logical operators to compare the results of multiple conditional expressions.

Operator	Description	Example
<code>and</code>	Returns True if both conditional expressions are True	<code>(age >= 5) and (height >= 54)</code>
<code>or</code>	Returns True if either one of the conditional expressions are True	<code>(color == blue) or (color == purple)</code>
<code>not</code>	Negates the result of the conditional expression	<code>(can_read) and (not dyslexic)</code>

Table 13-2: Python Logical Operators

1.4 IDENTITY OPERATORS

Table 13-3 lists Python's identity operators. Use an identity operator if you need to verify if two variables are actually the same object (i.e., refer to the same location in memory).

Operator	Description	Example
<code>is</code>	Returns True if both variables are the same object	<code>object_in_queue is mike</code>
<code>is not</code>	Returns True if both variables are not the same object	<code>mike is not sally</code>

Table 13-3: Python Identity Operators

1.5 MEMBERSHIP OPERATORS

Table 13-4 lists Python's membership operators. Use a membership operator to check if a sequence is present in another sequence. The operators are most often used to check if a substring is present within a larger string sequence.

Operator	Description	Example
<code>in</code>	Returns True if the sequence is present in the target sequence	<code>"llo" in "hello"</code>
<code>not in</code>	Returns True if the sequence is not in the target sequence	<code>"llo" not in ["hello", "world"]</code>

Table 13-4: Python Membership Operators

1.6 TYPE CHECKING

Table 13-5 lists Python’s built-in functions you can use in conditional expressions to verify an object’s type. This comes in handy when you want to enforce argument types passed to function and method calls.

Built-In	Description	Example
<code>isinstance()</code>	Returns True if object is instance of specified type	<code>isinstance(json_string, str)</code>
<code>issubclass()</code>	Returns True if object is instance of subclass type	<code>issubclass(employee, Person)</code>

Table 13-5: Python Type Checking Built-In Functions

Referring to table 13-5 — You will use the `isinstance()` built-in function quite often. I will wait and cover the `issubclass()` built-in function later in the book in the chapters on object-oriented programming.

QUICK REVIEW

Python’s control flow statements take action based on the results of conditional expressions. You can build conditional expressions from Python’s comparison, logical, identity, and membership operators. Two additional built-in functions you’ll find helpful include `isinstance()` and `issubclass()`.

Various types of Python objects are considered inherently *truthy* or *falsy*. At minimum, you should memorize that Python’s boolean literal `True` and the numeric value `1` always evaluate to *true*. Conversely, the boolean literal `False` and the numeric value `0` always evaluate to *false*. As a rule, non-empty data structures and strings evaluate to *true*, while empty data structures and strings evaluate to *false*.

Always verify the *truthy* or *falsy* behavior of an object before relying on its behavior in your code.

2 BRANCHING STATEMENTS

Branching statements enable you to specify which path of execution to follow based on the results of a conditional expression. Python provides two types of branching statements: `if/elif/else` and `match`. Let’s start with the basic `if` statement.

2.1 IF STATEMENT

The `if` statement is used to select a path of execution based on the results of a conditional expression. A conditional expression, regardless of how complicated, will always evaluate to either *true* or *false*. Figure 13-2 provides a conceptual illustration of an `if` statement.

Referring to figure 13-2 — A program’s path of execution proceeds until it reaches the `if` statement at which point the conditional expression is evaluated. If the conditional expression

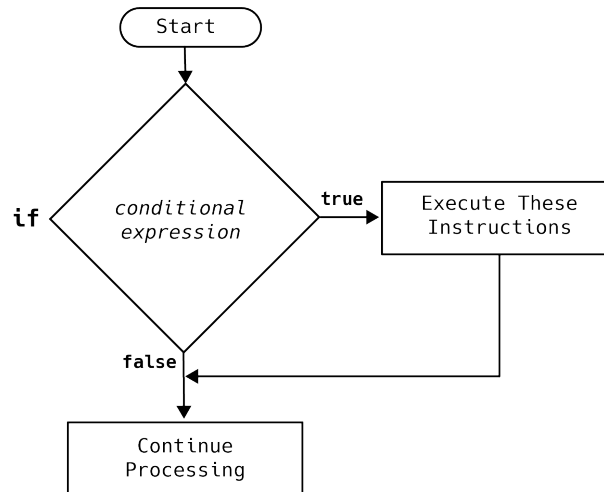


Figure 13-2: if Statement

evaluates to true, the program executes the set of statements specified by the `if` statement. If the conditional expression evaluates to false, the `if` statement's code is bypassed and processing continues. (Notice how I had to use an `if` statement to explain how the `if` statement works!)

OK, let's take a look at a simple `if` statement in action. Example 13.2 gives the code for a short program that checks the value of a datetime weekday name string and proceeds accordingly.

13.2 simple_if_statement.py

```

1  """Demonstrate the use of a simple if statement."""
2
3  from datetime import datetime
4
5  def main():
6      today = datetime.now()
7      print(f'Today\'s Date and Time in ISO Format: {today.isoformat()}')
8      print(f'Today\'s Weekday Number: {today.isoweekday()}')
9      print(f'Today\'s Weekday Name: {today.strftime("%A")}')
10
11     if (today.strftime('%A') == 'Saturday') or (today.strftime('%A') == 'Sunday'):
12         print('I love the weekends, don\'t you?')
13
14
15  if __name__ == '__main__':
16      main()
  
```

Referring to example 13.2 — Line 1 starts with an optional module-level doc comment that explains the purpose of the program. Line 3 imports Python's `datetime` module. The `main()` function defined starting on line 5 is where all the action takes place. On line 6, I call the `datetime.now()` function and assign the result to the variable named `today`. I then use the `today` variable in the print statements on lines 7 through 9 to demonstrate a few commonly-used functions of the `datetime` object.

Continuing with example 13.2 — The `if` statement on line 11 checks the value of the weekday name returned by the `datetime.strftime()` function and if it matches either the string 'Saturday' or 'Sunday' then the print statement on line 12 executes. If the weekday name is not 'Saturday' or 'Sunday' then line 12 is not executed and the program exits because it runs out of statements to execute. Figure 13-3 shows the results of running this program on a Saturday.

```

Sat Jan 13 06:57:55 EST 2024
~/dev/cst_with_python_1st_ed/chapter13/SimpleIfStatement (main)
[523:23] swodog@macos-mojave-testbed $ python3 simple_if_statement.py
Today's Date and Time in ISO Format: 2024-01-13T06:59:27.320541
Today's Weekday Number: 6
Today's Weekday Name: Saturday
I love the weekends, don't you?

```

Figure 13-3: Results of Running Example 13.2

2.1.1 USE PARENTHESES TO INDICATE OPERATOR PRECEDENCE

Again, referring to example 13.2 — Notice the conditional expression on line 12 consists of two subexpressions formed with the equality operator `==`. I then apply the logical or to those two subexpressions. I have also used parentheses to explicitly group the subexpressions. It is not strictly necessary to use parentheses as the operators have what is known as *precedence*. In this example, the comparison operators have a higher precedence than the logical operators, so the subexpressions that use the equality operator `==` are evaluated first followed by the application of the logical operator `or`.

I recommend you use parentheses to explicitly indicate your intentions in a complex conditional expression and do not rely on your memory of operator precedence. Doing so will make your code easier to read and understand, and more importantly, it will make it easier for you to figure out why your conditional expression isn't working as you intended.

Pro Tip: Use parentheses to explicitly indicate operator precedence. Doing so will reduce logical errors in complex conditional expressions while at the same time making your code easier to read and comprehend.

2.2 IF/ELSE STATEMENT

The `if` statement can be combined with an `else` clause to form an `if/else` statement as illustrated in figure 13-4. Whereas the simple `if` statement presents only one alternate path of execution, an `if/else` statement presents two. One path executes if the conditional expression evaluates to true, and the alternate path executes if the conditional expression evaluates to false. Example 13.3 builds upon the previous example and demonstrates the use of an `if/else` statement.

13.3 `if_else_statement.py`

```

1  """Demonstrate the use of an if/else statement."""
2
3  from datetime import datetime
4
5  def main():
6      today = datetime.now()
7      print(f'Today\'s Date and Time in ISO Format: {today.isoformat()}')
8      print(f'Today\'s Weekday Number: {today.isoweekday()}')
9      print(f'Today\'s Weekday Name: {today.strftime("%A")}')
10

```

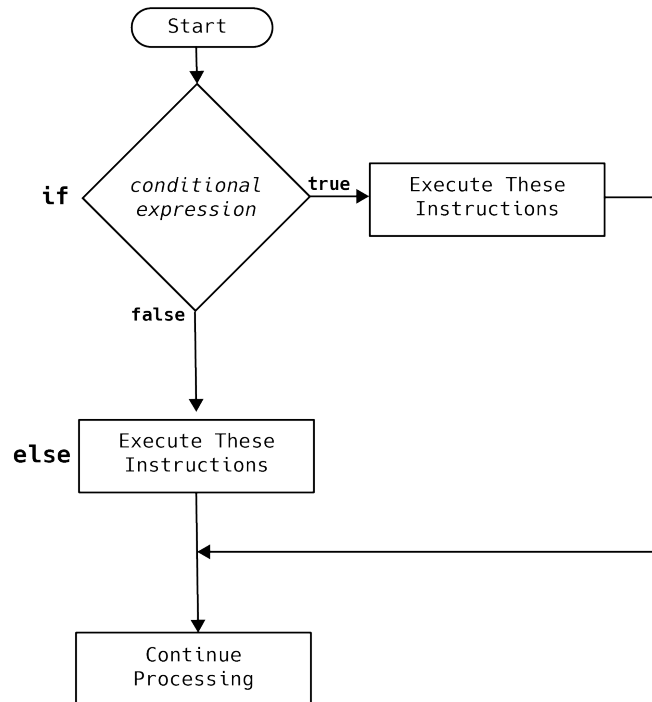



Figure 13-4: if/else Statement

```

11     if (today.strftime('%A') == 'Saturday') or (today.strftime('%A') == 'Sunday'):
12         print('I love the weekends, don\'t you?')
13     else:
14         print(f'Oh, too bad, it\'s {today.strftime("%A")} and you must go to work!')
15
16
17     if __name__ == '__main__':
18         main()
19

```

Referring to 13.3 — I’ve added an else clause on line 13. The if/else statement now reads like so: if the weekday name is either 'Saturday' or 'Sunday' execute line 12, else, execute line 14. It just happens to now be a beautiful Sunday morning as I write this, and figure 13-5 shows the results of running this program.

```

Sun Jan 14 07:47:06 EST 2024
~/dev/cst_with_python_1st_ed/chapter13/IfElseStatement (main)
[515:15] swodog@macos-mojave-testbed $ python3 if_else_statement.py
Today's Date and Time in ISO Format: 2024-01-14T07:47:09.258226
Today's Weekday Number: 7
Today's Weekday Name: Sunday
I love the weekends, don't you?

```

Figure 13-5: Results of Running Example 13.3

To test the else clause on a Sunday, I’ll need to make a slight modification to the code and manually set the date. Example 13.4 lists the modified code.

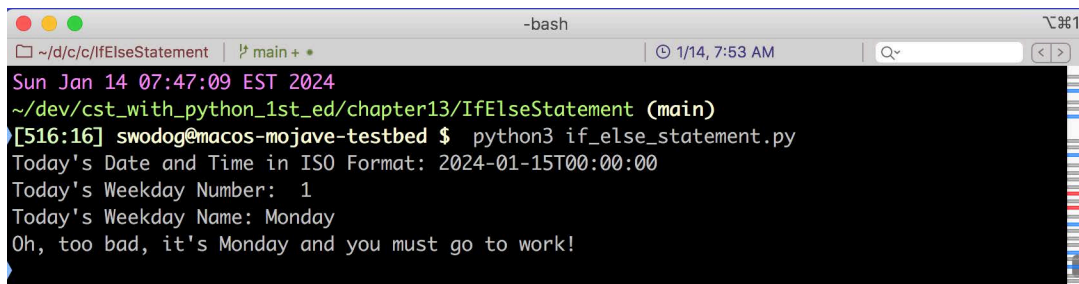
13.4 *if_else_statement.py (Mod 1)*

```

1  """Demonstrate the use of an if/else statement."""
2
3  from datetime import datetime
4
5  def main():
6      #today = datetime.now()
7      today = datetime(2024, 1, 15)
8      print(f'Today\'s Date and Time in ISO Format: {today.isoformat()}')
9      print(f'Today\'s Weekday Number: {today.isoweekday()}')
10     print(f'Today\'s Weekday Name: {today.strftime("%A")}')
11
12     if (today.strftime('%A') == 'Saturday') or (today.strftime('%A') == 'Sunday'):
13         print('I love the weekends, don\'t you?')
14     else:
15         print(f'Oh, too bad, it\'s {today.strftime("%A")} and you must go to work!')
16
17
18 if __name__ == '__main__':
19     main()
20

```

Referring to example 13.4 — I commented out line 6, which used the `datetime.now()` method to return the current date and time, and I am now setting a specific date by passing in year, month, and day arguments to the `datetime` constructor. This date, which is a Monday, will allow me to test the `else` clause as shown in figure 13-6.



```

Sun Jan 14 07:47:09 EST 2024
~/dev/cst_with_python_1st_ed/chapter13/IfElseStatement (main)
[516:16] swodog@macos-mojave-testbed $ python3 if_else_statement.py
Today's Date and Time in ISO Format: 2024-01-15T00:00:00
Today's Weekday Number: 1
Today's Weekday Name: Monday
Oh, too bad, it's Monday and you must go to work!

```

Figure 13-6: Results of Running Example 13.4

2.3 SINGLE VS. DOUBLE QUOTES

Before moving on, I'd like to comment on the use of single and double quotes in these last two example programs. The Python coding standard advises to be consistent with your use of quotation marks. As you are now well aware, you can use single or double quotation marks to indicate strings. However, you will encounter instances where a string enclosed in single quotation marks will need to include a single quotation mark, as in the case of contractions. (i.e., it's, I'm, you're, etc.) Since I have used single quotes for my strings in the `print()` functions, I must use an *escape sequence* to use a single quote within the single-quoted string. For example, on line 13 of the previous example I have the following `print()` statement:

```
print('I love the weekends, don\'t you?')
```

The escape sequence for a single quote is a backslash followed by the single-quote character `'\''`. Common escape sequences you'll find helpful include tab `'\t'`, line feed `'\n'`, and carriage return `'\r'`. These last two are often used together like so: `'\r\n'`. For a complete list of escape sequences see Appendix B — “Escape Sequences” on page 526.

2.4 IF/ELIF/ELSE STATEMENT

If you need to present more than two alternative execution paths you'll need to add one or more `elif` clauses to your `if` statement. Figure 13-7 gives the general structure of an `if/elif/else` statement.

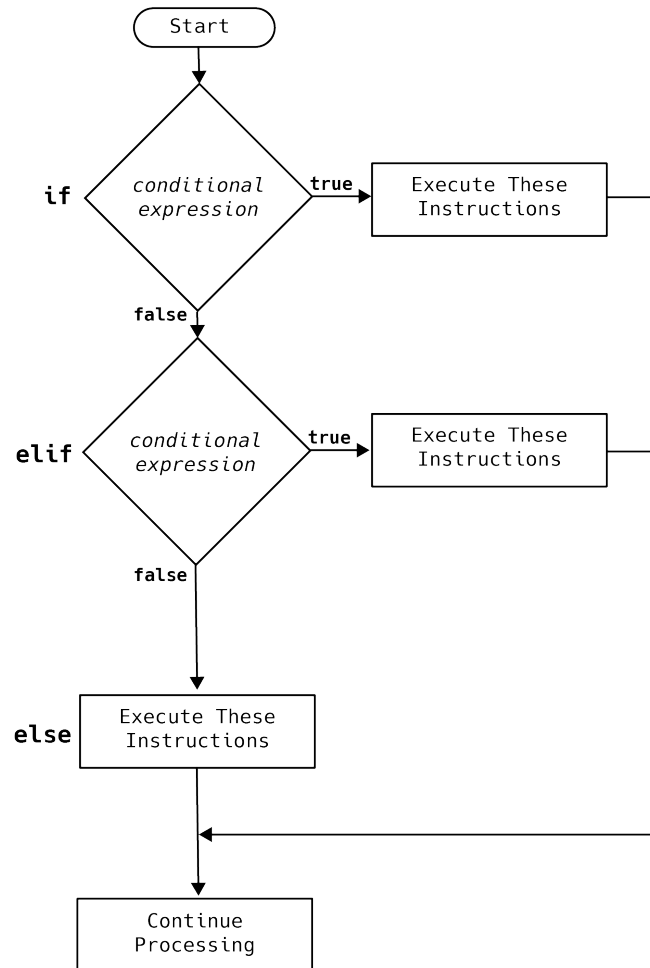


Figure 13-7: `if/elif/else` Statement

Referring to figure 13-7 — Note first that I'm only showing one `elif` clause above, but you can use as many `elif` clauses as you require. (However, after more than a handful of `elif` clauses I recommend using the `match` statement covered in the next section.) The `else` clause is optional and is used to provide a default alternate execution path. In other words, if the conditional expressions for the `if` and `elif` clauses evaluate to `false`, then the `else` clause is used to perform some type of action as is shown in example 13.5.

13.5 `if_elif_else_statement.py`

```

1  """Demonstrate the use of an if/elif/else statement."""
2
3  from datetime import datetime
4
5  def main():
6
7      try:

```

```

8
9     user_input = input("Enter yyyy mm dd: ").split()
10    year = int(user_input[0])
11    month = int(user_input[1])
12    day = int(user_input[2])
13
14    today = datetime(year, month, day)
15    weekday_name = today.strftime('%A')
16    print(f'Today\'s Date and Time in ISO Format: {today.isoformat()}')
17    print(f'Today\'s Weekday Number: {today.isoweekday()}')
18    print(f'Today\'s Weekday Name: {weekday_name}')
19
20    if (weekday_name == 'Saturday') or (weekday_name == 'Sunday'):
21        print(f'It\'s {weekday_name}! I love the weekends, don\'t you?')
22    elif weekday_name == 'Monday':
23        print('It\'s manic Monday -- I wish it were Sunday!')
24    elif weekday_name == 'Tuesday':
25        print(f'It\'s {weekday_name} -- I have class {weekday_name} evening.')
26    elif weekday_name == 'Wednesday':
27        print(f'I have a doctor\'s appointment on {weekday_name}!')
28    elif weekday_name == 'Thursday':
29        print(f'I devote {weekday_name}s to fasting.')
30    else:
31        print(f'It\'s {weekday_name}! The weekend is almost here!')
32    except Exception as e:
33        print(f'Problem processing date values. Please try again!: {e}')
34
35
36    if __name__ == '__main__':
37        main()
38

```

Referring to example 13.5 — When this program executes, it prompts the user to enter a date in yyyy mm dd format. The input string is split at the spaces by the `string.split()` method, and each element of the resulting array of strings is converted into an `int` on lines 10 through 12 and assigned to the corresponding year, month, and day variables from which a `datetime` object is created on line 14. Figure 13-8 shows the results of running this program.

```

Sun Jan 14 12:01:20 EST 2024
~/dev/cst_with_python_1st_ed/chapter13/IfElifElseStatement (main)
[533:33] swodog@macos-mojave-testbed $ python3 if_elif_else_statement.py
Enter yyyy mm dd: 2024 01 19
Today's Date and Time in ISO Format: 2024-01-19T00:00:00
Today's Weekday Number: 5
Today's Weekday Name: Friday
It's Friday! The weekend is almost here!

Sun Jan 14 12:01:42 EST 2024
~/dev/cst_with_python_1st_ed/chapter13/IfElifElseStatement (main)
[534:34] swodog@macos-mojave-testbed $ python3 if_elif_else_statement.py
Enter yyyy mm dd: 2024 2 4
Today's Date and Time in ISO Format: 2024-02-04T00:00:00
Today's Weekday Number: 7
Today's Weekday Name: Sunday
It's Sunday! I love the weekends, don't you?

```

Figure 13-8: Results of Running Example 13.5

2.5 MATCH STATEMENT

The match statement first appeared in Python 3.10 and can be used in place of an `if/elif/else` statement that contains multiple `elif` clauses. While there is nothing wrong with complex `if/elif/else` statements, the match statement provides a cleaner, more concise, and easier to read alternative. The general structure of a match statement is shown in figure 13-9.

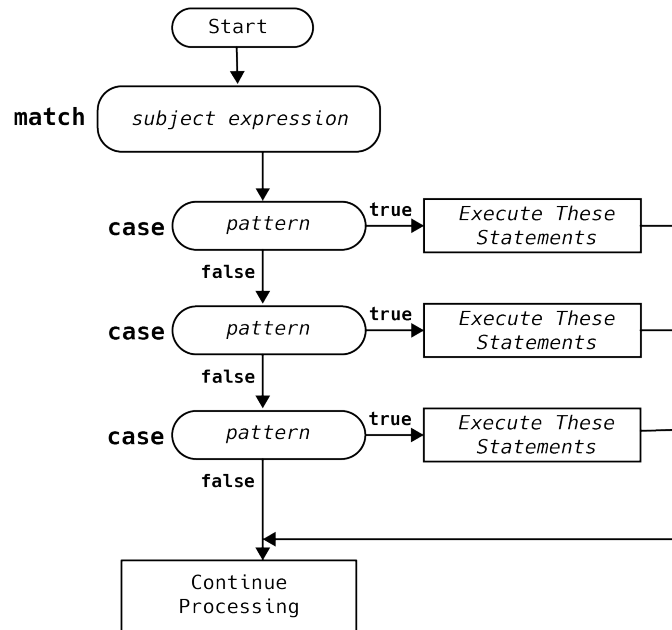


Figure 13-9: match Statement

Referring to figure 13-9 — The subject expression is not a conditional expression, rather, it is a subject that is compared against each of the case patterns. If the subject matches one of the case patterns, the code associated with that case is executed. You can also provide a default case that executes if no case patterns match the subject.

Example 13.6 employs a match statement to rework the previous example.

```

1  """Demonstrate the use of the match statement."""
2
3  from datetime import datetime
4
5  def main():
6
7      try:
8
9          user_input = input("Enter yyyy mm dd: ").split()
10         year = int(user_input[0])
11         month = int(user_input[1])
12         day = int(user_input[2])
13
14         today = datetime(year, month, day)
15         weekday_name = today.strftime('%A')
16         print(f'Today\'s Date and Time in ISO Format: {today.isoformat()}')
17         print(f'Today\'s Weekday Number: {today.isoweekday()}')
18         print(f'Today\'s Weekday Name: {weekday_name}')
19

```

13.6 match_statement.py

```

20     match weekday_name:
21         case 'Saturday' | 'Sunday':
22             print(f'It\'s {weekday_name}! I love the weekends, don\'t you?')
23         case 'Monday':
24             print('It\'s manic Monday -- I wish it were Sunday!')
25         case 'Tuesday':
26             print(f'It\'s {weekday_name} -- I have class {weekday_name}')
27         case 'Wednesday':
28             print(f'I have a doctor\'s appointment on {weekday_name}!')
29         case 'Thursday':
30             print(f'I devote {weekday_name}s to fasting.')
31         case _:
32             print(f'It\'s {weekday_name}! The weekend is almost here!')
33
34     except Exception as e:
35         print(f'Problem processing date values. Please try again!: {e}')
36
37
38 if __name__ == '__main__':
39     main()
40

```

Referring to example 13.6 — Starting on line 20, the match statement compares the *subject* `weekday_name`, which is a string in this example, against each case *pattern*, which also happen to be strings. Patterns appear between the case keywords and the colons ':'. Note that the first case specifies two patterns 'Saturday' | 'Sunday'. The '|' operator is used to present multiple patterns in one case. Figure 13-10 shows the results of running this program.

```

Sun Jan 21 08:57:00 EST 2024
~/dev/cst_with_python_1st_ed/chapter13/MatchStatement (main)
[519:19] swodog@macos-mojave-testbed $ python3 match_statement.py
Enter yyyy mm dd: 2024 1 21
Today's Date and Time in ISO Format: 2024-01-21T00:00:00
Today's Weekday Number: 7
Today's Weekday Name: Sunday
It's Sunday! I love the weekends, don't you?

Sun Jan 21 08:57:28 EST 2024
~/dev/cst_with_python_1st_ed/chapter13/MatchStatement (main)
[520:20] swodog@macos-mojave-testbed $ python3 match_statement.py
Enter yyyy mm dd: 2024 1 29
Today's Date and Time in ISO Format: 2024-01-29T00:00:00
Today's Weekday Number: 1
Today's Weekday Name: Monday
It's manic Monday -- I wish it were Sunday!

```

Figure 13-10: Results of Running Example 13.6

Referring to figure 13-10 — I've run the program twice with dates that fall on a Sunday and a Monday respectively. I'll leave it up to you to validate the remaining cases. Note that in this section I have presented only a tiny fraction of the match statement's capabilities.

QUICK REVIEW

Use an `if/elif/else` statement when you need to choose between alternate execution paths in your code. A simple `if` statement provides one alternate execution path. An `if/else` statement provides two alternate execution paths. An `if/elif/else` provides multiple alternate execution paths. You can use as many `elif` clauses as required, though too many tends to clutter the code.

Use the `match` statement as an alternative to complex `if/elif/else` statements. A `match` statement compares a subject expression against one or more case patterns. Multiple pattern choices can be combined into one case with the `'|'` operator.

3 LOOPING STATEMENTS

Looping statements allow you to repeatedly execute blocks of code. Python provides two types of looping statements: the `for` statement and the `while` statement. Let's begin with the `for` statement.

3.1 FOR STATEMENT

The `for` statement is used primarily to execute a designated block of code a set number of times. Each time a `for` statement executes its designated block of code it is referred to as an *iteration*. Figure 13-11 provides a diagram for a general `for` statement.

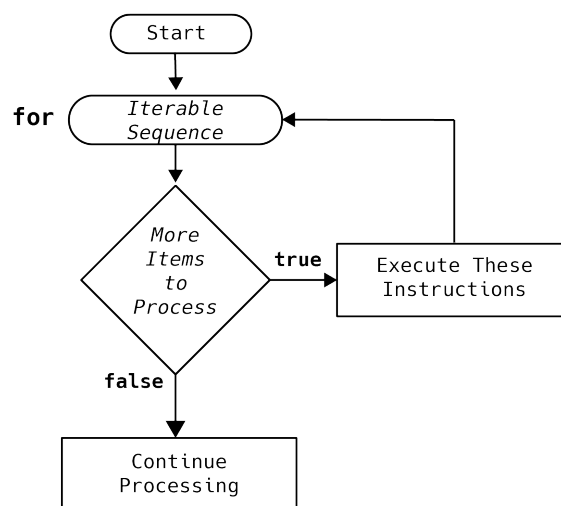


Figure 13-11: `for` Statement

Referring to figure 13-11 — The `for` statement processes an *iterable sequence*. Common iterable sequences include strings, lists, or a range of integer values. Example 13.7 shows a string being processed by a `for` loop.

```

1  """Demonstrate the use of a simple for statement."""
2
3  def main():
4      try:
5          input_string = input('Enter input string: ')

```

13.7 *simple_for_statement.py*

```

6     separator_char = input('Enter separator char: ')
7     for s in input_string:
8         print(f'{s}{separator_char}', end='')
9
10    except Exception as e:
11        print(f'Problem processing input string: {e}')
12
13
14    if __name__ == '__main__':
15        main()
16

```

Referring to example 13.7 — On line 5, this program prompts the user to enter a string with the help of the built-in `input()` function. Note that everything you type at the console is considered a string in Python. When you hit **Enter**, the string value returned by the `input()` function is assigned to the `input_string` variable. On line 6, another request is made for the user to enter a separator character. This could be anything the user types at the keyboard including multiple characters. (*Remember: Everything entered at the console is a string, whether it contains one character or multiple characters.*) The `for` statement on line 7 steps through each element of the `input_string` and executes the `print()` function on line 8, which uses a formatted string (a.k.a., an 'f' string) to print each character `s` from the `input_string` along with the `separator_char`. Figure 13-12 shows the results of running this program first with one space separator then with two spaces as the separator.

```

Sun Jan 21 12:00:35 EST 2024
~/dev/cst_with_python_1st_ed/chapter13/SimpleForStatement (main)
[550:50] swodog@macos-mojave-testbed $ python3 simple_for_statement.py
Enter input string: Hello World!
Enter separator char:
H e l l o   W o r l d !

Sun Jan 21 12:00:51 EST 2024
~/dev/cst_with_python_1st_ed/chapter13/SimpleForStatement (main)
[551:51] swodog@macos-mojave-testbed $ python3 simple_for_statement.py
Enter input string: Hello World!
Enter separator char:
H e l l o     W o r l d !

```

Figure 13-12: Results of Running Example 13.7 Using Blank Spaces as Separator Chars

Referring to figure 13-12 — I recommend running this program and studying the code. You can enter anything for the input string and separator char and obtain some pretty goofy results. You could totally use this little application to create strangely encoded messages. I'll leave that to you as an exercise.

Referring again to example 13.7 — The `for` statement on line 7 *iterates* over the string contained in the variable `input_string`. A string is a sequence of characters. You'll learn more about sequences in chapter 14. The `for` statement executes as many times as there are characters in the `input_string` sequence. Let's focus on the `for` statement as it appears on line 7:

```

for s in input_string:
    print(f'{s}{separator_char}', end='')

```

You can read this as: "for each element `s` in the sequence `input_string`, print the element `s` followed by the string `separator_char` to the console."

I just happened to use the letter 's' to denote each element in the `input_string`. I could have used any letter, like 'c' for example:

```
for c in input_string:
    print(f'{c}{separator_char}', end='')
```

As you'll learn later, a sequence could be a sequence of just about anything and what you choose to name the sequence (in this case I named the variable `input_string`) and each element of the sequence processed by the `for` statement (`s`, `c`, or whatever) is entirely up to you.

3.1.1 PROCESSING A RANGE OF NUMBERS

Sometimes, you want to execute a block of code for n amount of times. You can specify the n with the help of the built-in `range()` function. Let's look first at a short program that prints five integers to the console.

13.8 *for_five_times.py*

```
1  """Demonstrate built-in range() function."""
2
3  def main():
4      for i in range(5):
5          print(f'{i} ', end='')
6
7  if __name__ == '__main__':
8      main()
9
```

Referring to example 13.8 — The `for` statement on line 4 employs the built-in `range()` function to create an iterable sequence of integers 0 through 4. Figure 13-13 shows the results of running this program.



```

Sat Jan 27 08:23:37 EST 2024
~/dev/cst_with_python_1st_ed/chapter13/ForFiveTimes (main)
[523:23] swodog@macos-mojave-testbed $ python3 for_five_times.py
0 1 2 3 4

```

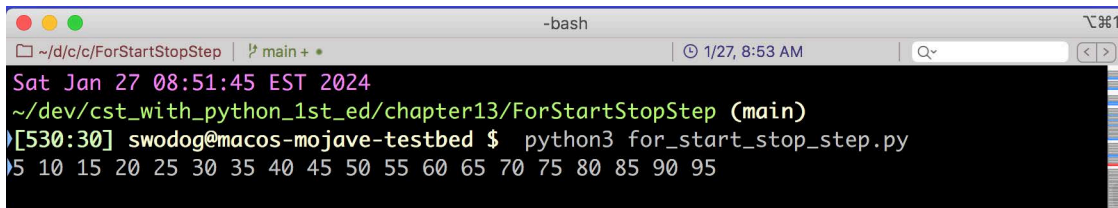
Figure 13-13: Results of Running Example 13.8

Referring again to example 13.8 — Actually, the `range()` function is not a function; it's an *immutable sequence type*. You'll learn more about these in chapter 8. In this example, I supplied `range()` with a stop value of 5. Given only a stop value n , `range(n)` will generate a sequence of integers from 0 to $n-1$, as you can see from the program's output in figure 13-13 above. You can more precisely control the sequence values by supplying `range()` with *start*, *stop*, and *step* values as demonstrated in example 13.9.

13.9 *for_start_stop_step.py*

```
1  """Demonstrate for statement with range(start, stop, step)."""
2
3  def main():
4      for i in range(5, 100, 5):
5          print(f'{i} ', end='')
6
7  if __name__ == '__main__':
8      main()
9
```

Referring to example 13.9 — The `range()` is now specified to start at 5, stop at 100, and step 5 units with each iteration. Figure 13-14 shows the results of running this program.



```

Sat Jan 27 08:51:45 EST 2024
~/dev/cst_with_python_1st_ed/chapter13/ForStartStopStep (main)
[530:30] swodog@macos-mojave-testbed $ python3 for_start_stop_step.py
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95

```

Figure 13-14: Results of Running Example 13.9

3.1.2 FOR LOOP WITH ELSE CLAUSE

You can add an `else` clause to a `for` loop. You would do this when you want to execute a block of code when the `for` loop completes execution. Example 13.10 provides a short program showing the `for/else` in action.

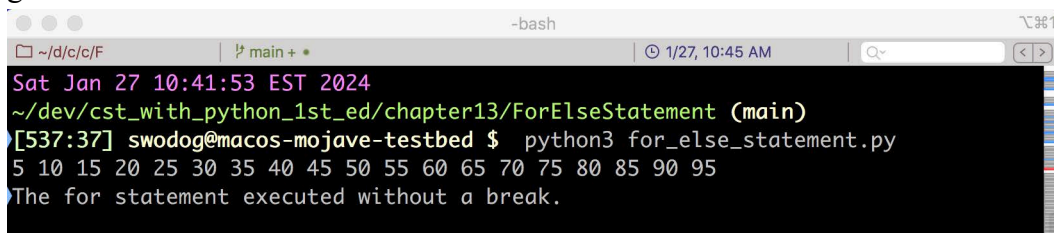
13.10

```

1  """Demonstrate the use of the for/else statement."""
2
3  def main():
4      for i in range(5, 100, 5):
5          print(f'{i} ', end='')
6      else:
7          print('\r\nThe for statement executed without a break.')
8
9
10 if __name__ == '__main__':
11     main()
12

```

Referring to example 13.10 — If the `for` statement runs to completion, the `else` clause on line 6 executes, which results in the execution of line 7. Figure 13-15 shows the results of running this program.



```

Sat Jan 27 10:41:53 EST 2024
~/dev/cst_with_python_1st_ed/chapter13/ForElseStatement (main)
[537:37] swodog@macos-mojave-testbed $ python3 for_else_statement.py
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95
The for statement executed without a break.

```

Figure 13-15: Results of Running Example 13.10

At this point, you will rightly be wondering, "When might a `for` loop not execute to completion?" Good question. If the `for` loop encounters a `break` statement during processing, the `for` loop is terminated, and the `else` clause is skipped. A simple example will suffice to demonstrate a typical search scenario as example 13.11 shows.

13.11 *for_break_else_demo.py*

```

1  """Demonstrate the use of break statement within a for loop."""
2
3  def main():
4      input_text = input('Enter Text To Be Searched: ').split()
5      search_string = input('Enter Search String: ')
6

```

```

7     for s in input_text:
8         if search_string in s:
9             print('Found it!')
10            break
11        else:
12            print(f'Search string \"{search_string}\" not found.')
13
14
15 if __name__ == '__main__':
16     main()
17

```

Referring to example 13.11 — This program prompts the user to enter some text to be searched followed by search string. The text to be searched is split on the spaces contained within the string, which is the default delimiter character for the `string.split()` method. The `for` statement on line 7 iterates over the `input_text` variable which represents an iterable list of strings. The `if` statement on line 8 checks to see if the `search_string` is contained within 's'. If a match is made, the `print()` statement on line 9 executes followed by the `break` statement on line 10, which exits the `for` loop and skips the `else` clause. If the `for` loop manages to iterate through all the strings contained within `input_text` and fails to find a match, the `else` clause executes as shown in figure 13-16.

```

Sat Jan 27 11:07:46 EST 2024
~/dev/cst_with_python_1st_ed/chapter13/ForBreakElseDemo (main)
[544:44] swodog@macos-mojave-testbed $ python3 for_break_else_demo.py
Enter Text To Be Searched: Oh, I want a peanut butter sandwich!
Enter Search String: w
Found it!

Sat Jan 27 11:08:37 EST 2024
~/dev/cst_with_python_1st_ed/chapter13/ForBreakElseDemo (main)
[545:45] swodog@macos-mojave-testbed $ python3 for_break_else_demo.py
Enter Text To Be Searched: Oh, I want a peanut butter sandwich!
Enter Search String: Hello World!
Search string "Hello World!" not found.

```

Figure 13-16: Results of Running Example 13.11

3.2 WHILE STATEMENT

Choose a `while` statement when you need to repeat a block of code as long as a conditional expression evaluates to `true`. Figure 13-17 illustrates a general `while` statement.

Up until now, all the examples in this chapter have executed once and exited right away. Sometimes this is the behavior you desire in a program. Most often, however, you want to perform some action until the user decides they want to quit. Example 13.12 reworks the earlier search example with the help of a `while` statement.

13.12 while_statement

```

1     """Demonstrate the use of the while statement."""
2
3     def main():
4         input_text = ''
5         while not ('quit' in input_text):
6             if not (input_text == ''):

```

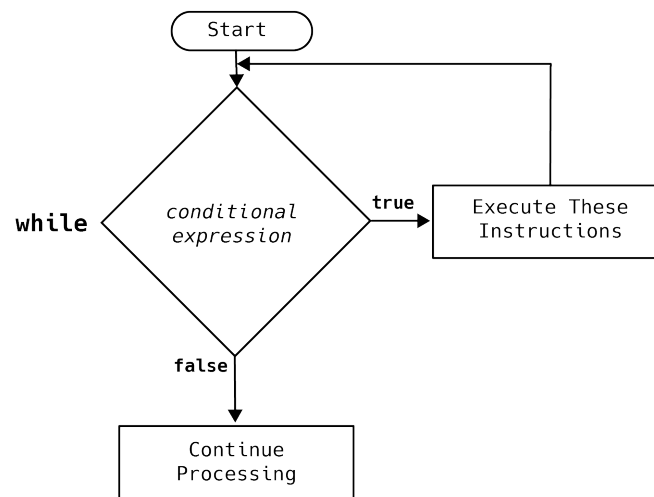


Figure 13-17: while Statement

```

7         search_string = input('Enter Search String: ')
8
9         for s in input_text:
10            if search_string in s:
11                print('Found it!')
12                break
13            else:
14                print(f'Search string \"{search_string}\" not found.')
15
16         input_text = input('Enter Text To Be Searched: ').split()
17
18
19     if __name__ == '__main__':
20         main()
21

```

Referring to example 13.12 — The variable `input_text` is initialized with an empty string. The `while` statement begins on line 5 and is read, "*while not the string 'quit' in input_text*". I have used parentheses to indicate my intentions with regards to operator precedence. Note that I could have also written the conditional expression like so:

```
while 'quit' not in input_text:
```

In this case, I believe this is the most natural way to construct the conditional expression. No parentheses are required because the operator precedence is obvious. But this illustrates a good point. Sometimes, in the heat of battle, the code you write is not necessarily your best, most elegant code. The act of coding mimics that of writing. I always consider my first attempt a rough draft, and several revisions may be required to ultimately produce a polished work.

Again referring to example 13.12 — First, note how the code is indented. When you first start to write complex Python programs, you will struggle with indenting, but indenting matters in Python. All code indented underneath the `while` statement belongs to the `while` statement. The purpose of the `if` statement on line 6 is to avoid having the user enter a search string when the `input_text` variable is initially set to the empty string. Everything indented under the `if` statement belongs to the `if` statement. The user doesn't enter the text to be searched until line 16,

which is the last statement within the `while` loop. If the user enters anything other than 'quit', the `while` loop continues to execute. Figure 13-18 shows the results of running this program.

```

Sun Jan 28 06:25:46 EST 2024
~/dev/cst_with_python_1st_ed/chapter13/WhileStatement (main)
[526:26] swodog@macos-mojave-testbed $ python3 while_statement.py
Enter Text To Be Searched: Oh, I love Python! Do you love Python, too?
Enter Search String: schmoogle
Search string "schmoogle" not found.
Enter Text To Be Searched: Oh, I love Python! Do you love Python, too?
Enter Search String: buster
Search string "buster" not found.
Enter Text To Be Searched: Oh, I love Python! Do you love Python, too?
Enter Search String: love
Found it!
Enter Text To Be Searched: quit
Sun Jan 28 06:40:24 EST 2024
~/dev/cst_with_python_1st_ed/chapter13/WhileStatement (main)
[527:27] swodog@macos-mojave-testbed $

```

Figure 13-18: Results of Running Example 13.12

3.2.1 WHILE/ELSE STATEMENT

You can also add an `else` clause to a `while` statement. The `else` clause executes when the `while` statement's conditional expression no longer holds true. Example 13.13 adds an `else` clause to the previous example.

13.13 *while_else_statement.py*

```

1  """Demonstrate the use of the while/else statement."""
2
3  def main():
4      input_text = ''
5      while 'quit' not in input_text:
6          if not (input_text == ''):
7              search_string = input('Enter Search String: ')
8
9              for s in input_text:
10                 if search_string in s:
11                     print('Found it!')
12                     break
13                 else:
14                     print(f'Search string \"{search_string}\" not found.')
15
16                 input_text = input('Enter Text To Be Searched: ').split()
17             else:
18                 print('Thank you for playing the search game!')
19
20
21 if __name__ == '__main__':
22     main()
23

```

Referring to example 13.13 — On line 17, I have added an `else` clause which executes when the user enters 'quit' to exit program. Figure 13-19 shows the results of running this program.

```

Sun Jan 28 07:25:45 EST 2024
~/dev/cst_with_python_1st_ed/chapter13/WhileElseStatement (main)
[535:35] swodog@macos-mojave-testbed $ python3 while_else_statement.py
Enter Text To Be Searched: Four score and seven years ago
Enter Search String: five
Search string "five" not found.
Enter Text To Be Searched: Four score and seven years ago
Enter Search String: and
Found it!
Enter Text To Be Searched: quit
Thank you for playing the search game!

Sun Jan 28 07:28:45 EST 2024
~/dev/cst_with_python_1st_ed/chapter13/WhileElseStatement (main)
[536:36] swodog@macos-mojave-testbed $ █

```

Figure 13-19: Results of Running Example 13.13

QUICK REVIEW

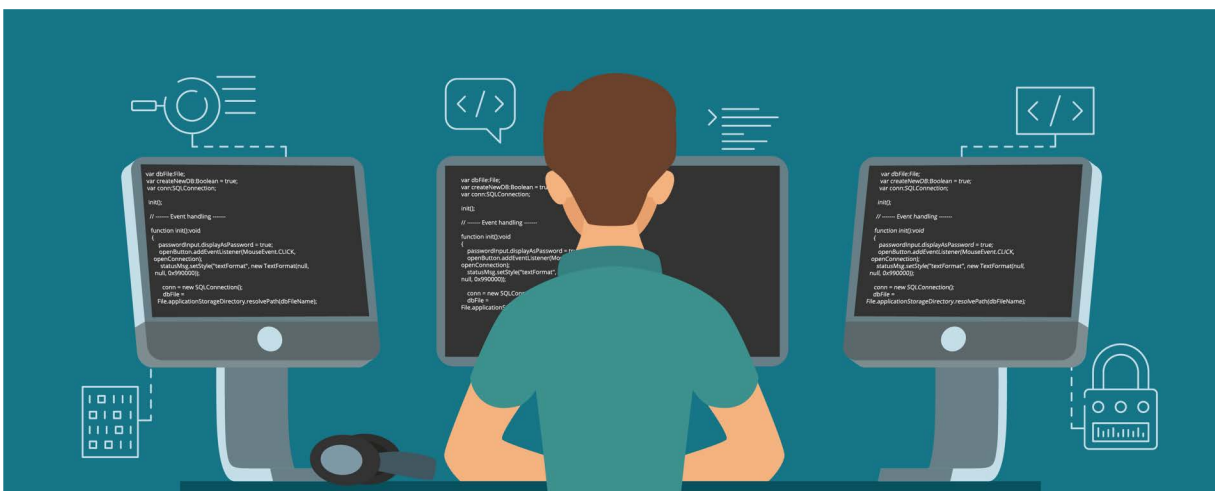
Python's looping statements include the `for` and `while` statements. Choose a `for` statement when you need to process elements contained within a sequence or to repeat a code block for a set amount of time.

Use the built-in `range()` function if you need to generate a list of integers.

Choose a `while` statement when you need to execute a block of code based on the results of a conditional expression.

Both the `for` and `while` statements support the use of an `else` clause. When used with a `for` statement, the `else` clause executes only if the `for` statement runs to completion. The `else` clause is skipped if the `for` statement exits early via a `break` statement.

When used with a `while` statement, the `else` clause executes when the conditional expression evaluates to false.



4 BRANCHING AND LOOPING STATEMENTS QUICK REFERENCE

Table 13-6 provided a quick reference guide for the control flow statements discussed in this chapter. If you're new to programming, you'll find it helpful to print the quick reference guide and keep it close at hand. Better yet, add it to your Engineer's Notebook.

Statement	Diagram	Usage
Branching Statements		
if/elif/else	<pre> graph TD Start([Start]) --> D1{conditional expression} D1 -- true --> I1[Execute These Instructions] D1 -- false --> D2{conditional expression} D2 -- true --> I2[Execute These Instructions] D2 -- false --> I3[Execute These Instructions] I1 --> CP[Continue Processing] I2 --> CP I3 --> CP </pre>	<p>Provides alternative execution paths based on results of conditional expression.</p> <p>Use a simple <code>if</code> statement if you need one alternative execution path.</p> <p>Use an <code>if/else</code> statement if you need two alternative execution paths.</p> <p>Use an <code>if/elif/else</code> statement if you need two or more alternative execution paths.</p>
match	<pre> graph TD Start([Start]) --> S[subject expression] S --> D1{pattern} D1 -- true --> I1[Execute These Statements] D1 -- false --> D2{pattern} D2 -- true --> I2[Execute These Statements] D2 -- false --> D3{pattern} D3 -- true --> I3[Execute These Statements] D3 -- false --> CP[Continue Processing] I1 --> CP I2 --> CP I3 --> CP </pre>	<p>Provides alternative execution paths based on the evaluation of a subject expression against one or more case patterns.</p> <p>Favor the <code>match</code> statement over the use of a complex <code>if/elif/else</code> statement for improved readability</p>

Table 13-6: Branching and Looping Statements Quick Reference

Statement	Diagram	Usage
Looping Statements		
for	<pre> graph TD Start([Start]) --> IS([Iterable Sequence]) IS --> MIP{More Items to Process} MIP -- true --> ETI[Execute These Instructions] ETI --> IS MIP -- false --> CP[Continue Processing] </pre>	<p>Provides repeated execution for set number of iterations. Select when you need to process elements in a sequence or execute a block of code n number of times.</p> <p>An optional <code>else</code> clause will execute if the <code>for</code> statement runs to completion. The <code>else</code> clause is skipped if the <code>for</code> loop exits early due to a <code>break</code> statement.</p>
while	<pre> graph TD Start([Start]) --> CE{conditional expression} CE -- true --> ETI[Execute These Instructions] ETI --> CE CE -- false --> CP[Continue Processing] </pre>	<p>Executes a block of code repeatedly if the conditional expression evaluates to true.</p> <p>An optional <code>else</code> clause will execute when the <code>while</code> statement's conditional expression evaluates to false.</p>

Table 13-6: Branching and Looping Statements Quick Reference (Continued)

SUMMARY

Python's control flow statements take action based on the results of conditional expressions. You can build conditional expressions from Python's comparison, logical, identity, and membership operators. Two additional built-in functions you'll find helpful include `isinstance()` and `issubclass()`.

Various types of Python objects are considered inherently `truthy` or `falsy`. At minimum, you should memorize that Python's boolean literal `True` and the numeric value `1` always evaluate to `true`. Conversely, the boolean literal `False` and the numeric value `0` always evaluate to `false`. As a rule, non-empty data structures and strings evaluate to `true`, while empty data structures and strings evaluate to `false`.

Always verify the `truthy` or `falsy` behavior of an object before relying on its behavior in your code.

Use an `if/elif/else` statement when you need to choose between alternate execution paths in your code. A simple `if` statement provides one alternate execution path. An `if/else` statement provides two alternate execution paths. An `if/elif/else` provides multiple alternate execution paths. You can use as many `elif` clauses as required, though too many tends to clutter the code.

Use the `match` statement as an alternative to complex `if/elif/else` statements. A `match` statement compares a subject expression against one or more case patterns. Multiple pattern choices can be combined into one case with the `|` operator.

Python's looping statements include the `for` and `while` statements. Choose a `for` statement when you need to process elements contained within a sequence or to repeat a code block for a set amount of time.

Use the built-in `range()` function if you need to generate a list of integers.

Choose a `while` statement when you need to execute a block of code based on the results of a conditional expression.

Both the `for` and `while` statements support the use of an `else` clause. If used with a `for` statement, the `else` clause executes only if the `for` statement runs to completion. The `else` clause is skipped if the `for` statement exits early via a `break` statement.

If used with a `while` statement, the `else` clause executes when the conditional expression evaluates to `false`.

SKILL-BUILDING EXERCISES

1. **Research:** Search the Internet for more information about Python conditional expressions. Study the formulation of complex conditional expressions.
2. **Research:** Create a table in your Engineers Notebook that lists the Python operators in order of precedence.
3. **Research:** Search the Internet for examples of complex `if/elif/else` statements in action. A good place to start would be GitHub.
4. **Research:** Dive deeper into the use of the `match` statement for pattern matching.
5. **Engineers Notebook Preparation:** Print the Branching and Looping Statements Quick Reference Guide provided in this chapter and copy or insert it into your Engineers Notebook.

SUGGESTED PROJECTS

1. **Hands On Coding:** Enter and run all the examples listed in this chapter. Study each line of code to ensure you understand what it's doing.
2. **Validate All Cases:** Run example 13.6 and enter dates that test all `match` statement cases.

SELF-TEST QUESTIONS

1. Which branching statements does Python support?
2. What's the primary difference between an `if/elif/else` statement and a `match` statement.
3. How many `elif` clauses can an `if` statement support?
4. When does an `if` statement's `else` clause execute?
5. When would you select a `match` statement over an `if/elif/else` statement?
6. Which looping statements does Python support?
7. Which looping statement would you use to iterate over elements contained within a sequence?
8. What's the built-in `range()` function used for?
9. When does a `for` statement's `else` clause execute?
10. What's the purpose of a `break` statement?
11. When does a `while` statement's `else` clause execute?

REFERENCES

Python Documentation, Compound Statements, https://docs.python.org/3/reference/compound_stmts.html#

Python Documentation, `range()` Function, <https://docs.python.org/3/library/functions.html#func-range>

W3 School Documentation, Python Operators, https://www.w3schools.com/python/python_operators.asp

PEP 622, Structural Pattern Matching, <https://peps.python.org/pep-0622/>

Martin McBride, Medium.com, The Hidden Traps of Python Truthy Values, <https://medium.com/geekculture/the-hidden-traps-of-python-truthy-values-51224e0413ab>

NOTES
