Selection: if statement

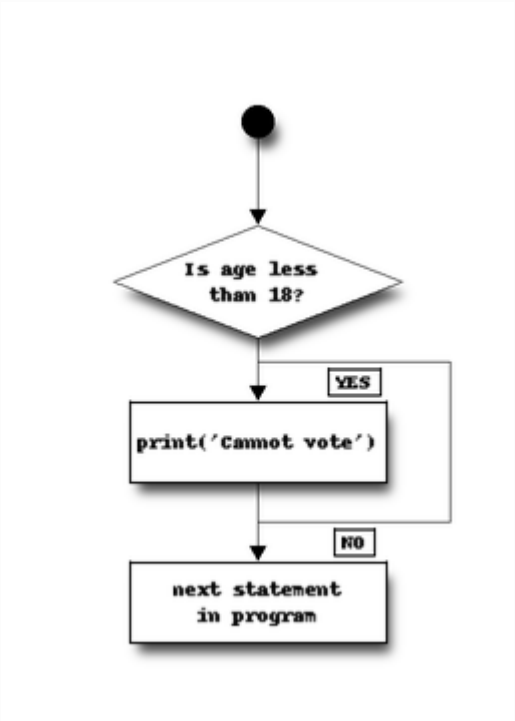
People make decisions on a daily basis. What should I have for lunch? What should I do this weekend? Every time you make a decision you base it on some criterion. For example, you might decide what to have for lunch based on your mood at the time, or whether you are on some kind of diet. After making this decision, you act on it. Thus decision-making is a two step process – first deciding what to do based on a criterion, and secondly taking an action.

Decision-making by a computer is based on the same two-step process. In Python, decisions are made with the if statement, also known as the selection statement. When processing an if statement, the computer first evaluates some criterion or condition. If it is met, the specified action is performed. Here is the syntax for the if statement:

```
if condition:
    if_body
```

When it reaches an if statement, the computer only executes the body of the statement only if the condition is true. Here is an example in Python, with a corresponding flowchart:

```
if age < 18:
    print("Cannot vote")
```



As we can see from the flowchart, the instructions in the if body are only executed if the condition is met (i.e. if it is true). If the condition is not met (i.e. false), the instructions in the ifbody are skipped.

Relational operators

Many if statements compare two values in order to make a decision. In the last example, we compared the variable age to the integer 18 to test if age less than 18. We used the operator <for the comparison. This operator is one of the relational operators that can be used in Python. The table below shows Python's relational operators.

Operator	Description	Example
==	equal to	if (age == 18)
!=	not equal to	if (score != 10)
>	greater than	if (num_people > 50)
<	less than	if (price < 25)
>=	greater than or equal to	if (total >= 50)
<=	less than or equal to	if (value <= 30)

Note that the condition statement can either be true or false. Also note that the operator for equality is == – a double equals sign. Remember that =, the single equals sign, is the assignment operator. If we accidentally use = when we mean ==, we are likely to get a syntax error:

```
>>> if choice = 3:
File "<stdin>", line 1
  if choice = 3:
            ^
SyntaxError: invalid syntax
```

This is correct:

```
if choice == 3:
    print("Thank you for using this program.")
```

Note

in some languages, an assignment statement *is* a valid conditional expression: it is evaluated as true if the assignment is executed successfully, and as false if it is not. In such languages, it is easier to use the wrong operator by accident and not notice!

Value vs identity

So far, we have only compared integers in our examples. We can also use any of the above relational operators to compare floating-point numbers, strings and many other types:

```
# we can compare the values of strings
if name == "Jane":
    print("Hello, Jane!")

# ... or floats
if size < 10.5:
    print(size)
```

When comparing variables using ==, we are doing a *value* comparison: we are checking whether the two variables have the same value. In contrast to this, we might want to know if two objects such as lists, dictionaries or custom objects that we have created ourselves are *the exact same object*. **This is a test of *identity*. Two objects might have identical contents, but be two different objects. We compare identity with the is operator**:

```
a = [1,2,3]
b = [1,2,3]

if a == b:
    print("These lists have the same value.")

if a is b:
    print("These lists are the same list.")
```

It is generally the case (with some caveats) that if two variables are the same object, they are also equal. The reverse is not true – two variables could be equal in value, but not the same object.

To test whether two objects are *not* the same object, we can use the is not operator:

```
if a is not b:
    print("a and b are not the same object.")
```

Note

In many cases, variables of built-in immutable types which have the same value will also be identical. In some cases this is because the Python interpreter saves memory (and comparison time) by representing multiple values which

are equal by the same object. You shouldn't rely on this behaviour and make value comparisons using is – if you want to compare values, always use ==.

Using indentation

In the examples which have appeared in this chapter so far, there has only been one statement appearing in the if body. Of course it is possible to have more than one statement there; for example:

```
if choice == 1:
    count += 1
    print("Thank you for using this program.")
print("Always print this.") # this is outside the if block
```

The interpreter will treat all the statements inside the indented block as one statement – it will process all the instructions in the block before moving on to the next instruction. This allows us to specify multiple instructions to be executed when the condition is met.

if is referred to as a *compound statement* in Python because it combines multiple other statements together. A compound statement comprises one or more *clauses*, each of which has a *header* (like if) and a *suite* (which is a list of statements, like the if body). The contents of the suite are delimited with indentation – we have to indent lines to the same level to put them in the same block.

The else clause

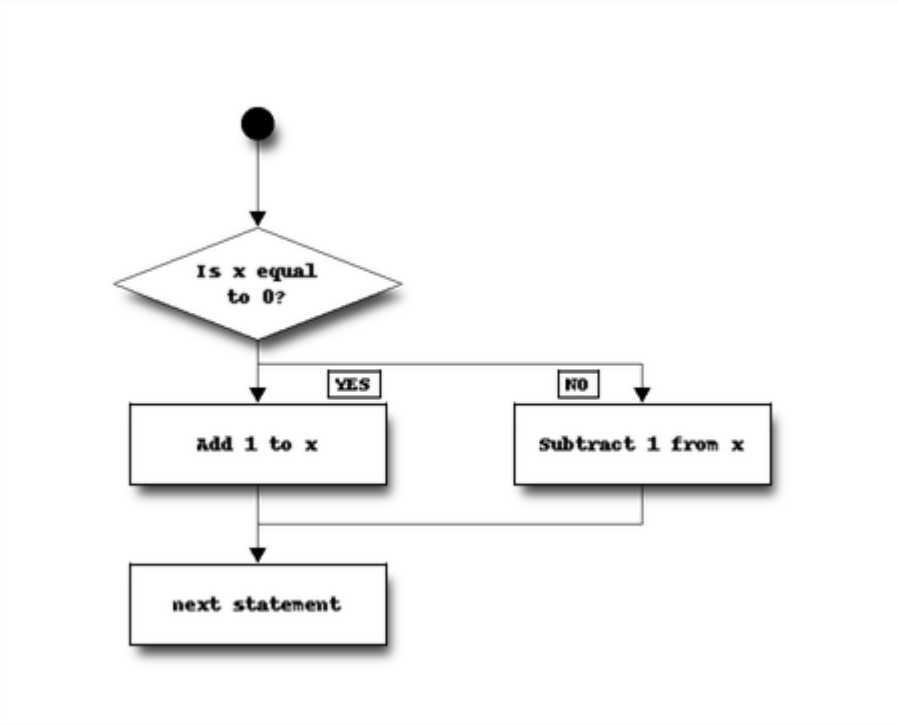
An optional part of an if statement is the else clause. It allows us to specify an alternative instruction (or set of instructions) to be executed if the condition is *not* met:

```
if condition:
    if_body
else:
    else_body
```

To put it another way, the computer will execute the if body if the condition is true, otherwise it will execute the else body. In the example below, the computer will add 1 to x if it is zero, otherwise it will subtract 1 from x:

```
if x == 0:
    x += 1
else:
    x -= 1
```

This flowchart represents the same statement:



The computer will execute one of the branches before proceeding to the next instruction.

Exercise 1

1. Which of these fragments are valid and invalid first lines of if statements? Explain why:

```
1. if (x > 4)
2. if x == 2
3. if (y =< 4)
4. if (y = 5)
5. if (3 <= a)
6. if (1 - 1)
7. if ((1 - 1) <= 0)
8. if (name == "James")
```

2. What is the output of the following code fragment? Explain why.

```
3. x = 2
4. if x > 3:
5.     print("This number")
6. print("is greater")
7. print("than 3.")
```

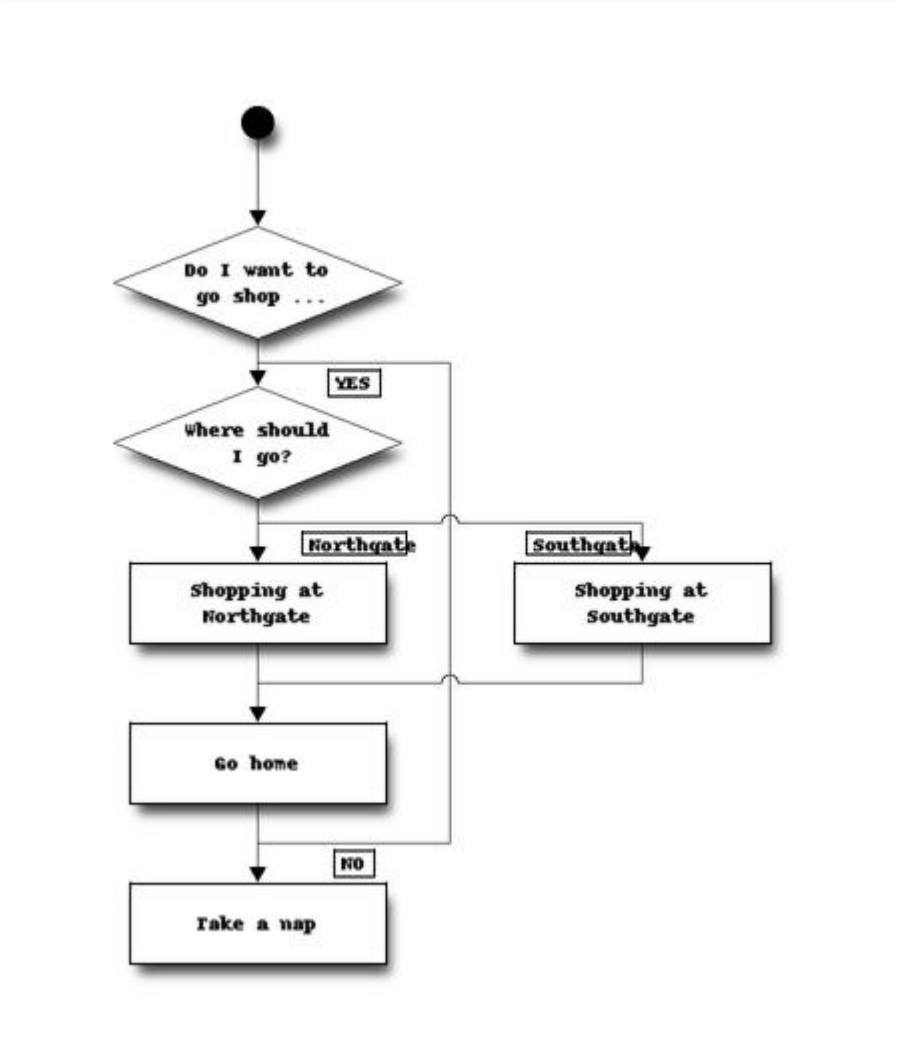
8. How can we simplify these code fragments?

```
1. if bool(a) == True:
2.     print("a is true")

3. if x > 50:
4.     b += 1
5.     a = 5
6. else:
7.     b -= 1
8.     a = 5
```

More on the if statement

Nested if statements

In some cases you may want one decision to depend on the result of an earlier decision. For example, you might only have to choose which shop to visit if you decide that you are going to do your shopping, or what to have for dinner after you have made a decision that you are hungry enough for dinner.



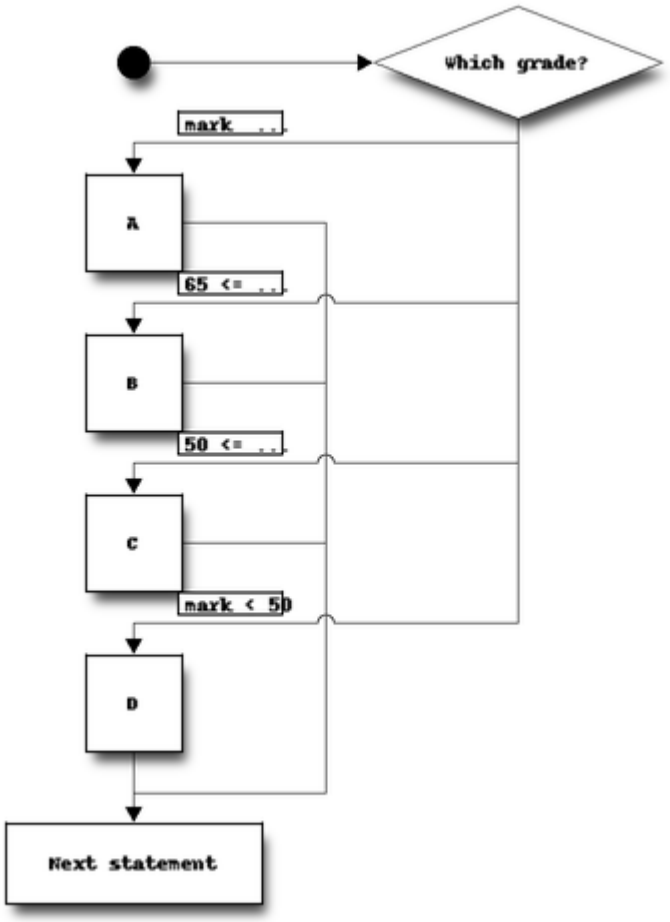
In Python this is equivalent to putting an if statement within the body of either the if or the else clause of another if statement. The following code fragment calculates the cost of sending a small parcel. The post office charges R5 for the first 300g, and R2 for every 100g thereafter (rounded up), up to a maximum weight of 1000g:

```
if weight <= 1000:
    if weight <= 300:
        cost = 5
    else:
        cost = 5 + 2 * round((weight - 300)/100)

    print("Your parcel will cost R%d." % cost)

else:
    print("Maximum weight for small parcel exceeded.")
    print("Use large parcel service instead.")
```

Note that the bodies of the outer if and else clauses are indented, and the bodies of the inner if and else clauses are indented one more time. It is important to keep track of indentation, so that each statement is in the correct block. It doesn't matter that there's an empty line between the last line of the inner if statement and the following print statement – they are still both part of the same block (the outer if body) because they are indented by the same amount. We can use empty lines (sparingly) to make our code more readable.

The elif clause and if ladders

The addition of the else keyword allows us to specify actions for the case in which the condition is false. However, there may be cases in which we would like to handle more than two alternatives. For example, here is a flowchart of a program which works out which grade should be assigned to a particular mark in a test:



We should be able to write a code fragment for this program using nested if statements. It might look something like this:

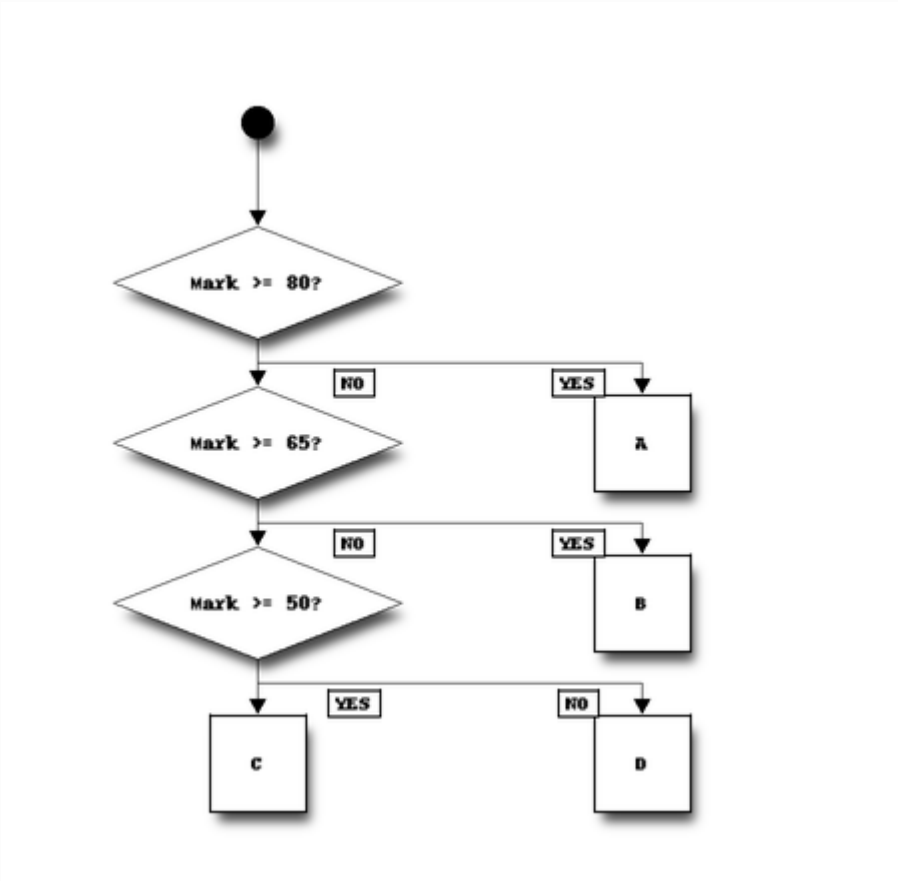
```
if mark >= 80:
    grade = A
else:
    if mark >= 65:
        grade = B
    else:
        if mark >= 50:
            grade = C
        else:
            grade = D
```

This code is a bit difficult to read. Every time we add a nested if, we have to increase the indentation, so all of our alternatives are indented differently. We can write this code more cleanly using elif clauses:

```
if mark >= 80:
```

```
    grade = A
elif mark >= 65:
    grade = B
elif mark >= 50:
    grade = C
else:
    grade = D
```

Now all the alternatives are clauses of one if statement, and are indented to the same level. This is called an *if ladder*. Here is a flowchart which more accurately represents this code:



The default (catch-all) condition is the else clause at the end of the statement. If none of the conditions specified earlier is matched, the actions in the else body will be executed. It is a good idea to include a final else clause in each ladder to make sure that we are covering all cases, especially if there's a possibility that the options will change in the future. Consider the following code fragment:

```
if course_code == "CSC":
    department_name = "Computer Science"
elif course_code == "MAM":
    department_name = "Mathematics and Applied Mathematics"
elif course_code == "STA":
```

```
    department_name = "Statistical Sciences"
else:
    department_name = None
    print("Unknown course code: %s" % course_code)

if department_name:
    print("Department: %s" % department_name)
```

What if we unexpectedly encounter an informatics course, which has a course code of "INF"? The catch-all else clause will be executed, and we will immediately see a printed message that this course code is unsupported. If the else clause were omitted, we might not have noticed that anything was wrong until we tried to use department_name and discovered that it had never been assigned a value. Including the else clause helps us to pick up potential errors caused by missing options early.

Boolean values, operators and expressions

The bool type

In Python there is a value type for variables which can either be true or false: the boolean type, bool. The true value is True and the false value is False. Python will implicitly convert any other value type to a boolean if we use it like a boolean, for example as a condition in an if statement. We will almost never have to cast values to bool explicitly. We also don't have to use the ==operator explicitly to check if a variable's value evaluates to True – we can use the variable name by itself as a condition:

```
name = "Jane"

# This is shorthand for checking if name evaluates to True:
if name:
    print("Hello, %s!" % name)

# It means the same thing as this:
if bool(name) == True:
    print("Hello, %s!" % name)

# This won't give us the answer we expect:
if name == True:
    print("Hello, %s!" % name)
```

Why won't the last if statement do what we expect? If we cast the string "Jane" to a boolean, it will be equal to True, but it isn't equal to True while it's still a string – so the condition in the last if statement will evaluate to False. This is why we should always use the shorthand syntax, as shown in the first statement – Python will then do the implicit cast for us.

Note

For historical reasons, the numbers 0 and 0.0 are actually equal to False and 1 and 1.0 are equal to True. They are not, however, identical objects – you can test this by comparing them with the is operator.

At the end of the previous chapter, we discussed how Python converts values to booleans implicitly. Remember that all non-zero numbers and all non-empty strings are True and zero and the empty string ("") are False. Other built-in data types that can be considered to be "empty" or "not empty" follow the same pattern.

Boolean operations

Decisions are often based on more than one factor. For example, you might decide to buy a shirt only if you like it AND it costs less than R100. Or you might decide to go out to eat tonight if you don't have anything in the fridge OR you don't feel like cooking. You can also alter conditions by negating them – for example you might only want to go to the concert tomorrow if it is NOT raining. Conditions which consist of simpler conditions joined together with AND, OR and NOT are referred to as *compound conditions*. These operators are known as *boolean operators*.

The and operator

The AND operator in Python is and. A compound expression made up of two subexpressions and the and operator is only true when *both* subexpressions are true:

```
if mark >= 50 and mark < 65:
    print("Grade B")
```

The compound condition is only true if the given mark is less than 50 *and* it is less than 65. The and operator works in the same way as its English counterpart. We can define the and operator formally with a truth table such as the one below. The table shows the truth value of a and b for every possible combination of subexpressions a and b. For example, if a is true and b is true, then a and b is true.

a	b	a and b
True	True	True
True	False	False

a	b	a and b
False	True	False
False	False	False

and is a binary operator so it must be given two operands. Each subexpression must be a valid complete expression:

```
# This is correct:
if (x > 3 and x < 300):
    x += 1

# This will give us a syntax error:
if (x > 3 and < 300): # < 300 is not a valid expression!
    x += 1
```

We can join three or more subexpressions with and – they will be evaluated from left to right:

condition_1 and condition_2 and condition_3 and condition_4

```
# is the same as
((condition_1 and condition_2) and condition_3) and condition_4
```

Note

for the special case of testing whether a number falls within a certain range, we don't have to use the and operator at all. Instead of writing mark >= 50 and mark < 65 we can simply write 50 <= mark < 65. This doesn't work in many other languages, but it's a useful feature of Python.

Short-circuit evaluation

Note that if a is false, the expression a and b is false whether b is true or not. The interpreter can take advantage of this to be more efficient: if it evaluates the first subexpression in an AND expression to be false, it does not bother to evaluate the second subexpression. We call and a *shortcut operator* or *short-circuit* operator because of this behaviour.

This behaviour doesn't just make the interpreter slightly faster – we can also use it to our advantage when writing programs. Consider this example:

```
if x > 0 and 1/x < 0.5:
    print("x is %f" % x)
```

What if x is zero? If the interpreter were to evaluate both of the subexpressions, we would get a divide by zero error. But because and is a short-circuit operator, the second subexpression will only be evaluated if the first subexpression is true. If x is zero, it will evaluate to false, and the second subexpression will not be evaluated at all.

We could also have used nested if statements, like this:

```
if x > 0:
    if 1/x < 0.5:
        print("x is %f" % x)
```

Using and instead is more compact and readable – especially if we have more than two conditions to check. These two snippets do the same thing:

```
if x != 0:
    if y != 0:
        if z != 0:
            print(1/(x*y*z))

if x != 0 and y != 0 and z != 0:
    print(1/(x*y*z))
```

This often comes in useful if we want to access an object's attribute or an element from a list or a dictionary, and we first want to check if it exists:

```
if hasattr(my_person, "name") and len(myperson.name) > 30:
    print("That's a long name, %s!" % myperson.name)

if i < len(mylist) and mylist[i] == 3:
    print("I found a 3!")

if key in mydict and mydict[key] == 3:
    print("I found a 3!")
```

The or operator

The OR operator in Python is or. A compound expression made up of two subexpressions and the or operator is true when *at least one* of the subexpressions is true. This means that it is only false in the case where both subexpressions are false, and is true for all other cases. This can be seen in the truth table below:

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

The following code fragment will print out a message if the given age is less than 0 *or* if it is more than 120:

```
if age < 0 or age > 120:
    print("Invalid age: %d" % age)
```

The interpreter also performs a short-circuit evaluation for or expressions. If it evaluates the first subexpression to be true, it will not bother to evaluate the second, because this is sufficient to determine that the whole expression is true.

The || operator is also binary:

```
# This is correct:
if x < 3 or x > 300:
    x += 1

# This will give us a syntax error:
if x < 3 or > 300: # > 300 is not a valid expression!
    x += 1

# This may not do what we expect:
if x == 2 or 3:
    print("x is 2 or 3")
```

The last example won't give us an error, because 3 is a valid subexpression – and since it is a non-zero number it evaluates to True. So the last if body will always execute, regardless of the value of x!

The not operator

The NOT operator, not in Python, is a unary operator: it only requires one operand. It is used to reverse an expression, as shown in the following truth table:

a	not a
True	False
False	True

The not operator can be used to write a less confusing expression. For example, consider the following example in which we want to check whether a string *doesn't* start with "A":

```
if name.startswith("A"):
    pass # a statement body can't be empty -- this is an instruction which does nothing.
else:
    print("'%s' doesn't start with A!" % s)

# That's a little clumsy -- let's use "not" instead!
if not name.startswith("A"):
    print("'%s' doesn't start with A!" % s)
```

Here are a few other examples:

```
# Do something if a flag is False:
if not my_flag:
    print("Hello!")

# This...
if not x == 5:
    x += 1

# ... is equivalent to this:
if x != 5:
    x += 1
```

Precedence rules for boolean expressions

Here is a table indicating the relative level of precedence for all the operators we have seen so far, including the arithmetic, relational and boolean operators.

Operators
() (highest)
**
*, /, %,//
+, -
<, <=, >, >= ==, !=
is, is not
not
and
or (lowest)

It is always a good idea to use brackets to clarify what we mean, even though we can rely on the order of precedence above. Brackets can make complex expressions in our code easier to read and understand, and reduce the opportunity for errors.

DeMorgan's law for manipulating boolean expressions

The not operator can make expressions more difficult to understand, especially if it is used multiple times. Try only to use the not operator where it makes sense to have it. Most people find it easier to read positive statements than negative ones. Sometimes we can use the opposite relational operator to avoid using the not operator, for example:

```
if not mark < 50:
    print("You passed")

# is the same as

if mark >= 50:
```

```
    print("You passed")
```

This table shows each relational operator and its opposite:

Operator	Opposite
==	!=
>	<=
<	>=

There are other ways to rewrite boolean expressions. The 19th-century logician DeMorgan proved two properties of negation that we can use.

Consider this example in English: *it is not both cool and rainy today.* Does this sentence mean the same thing as *it is not cool and not rainy today?* No, the first sentence says that both conditions are not true, but either one of them could be true. The correct equivalent sentence is *it is not cool or not rainy today.*

We have just used DeMorgan's law to distribute NOT over the first sentence. Formally, DeMorgan's laws state:

1. NOT (a AND b) = (NOT a) OR (NOT b)
2. NOT (a OR b) = (NOT a) AND (NOT b)

We can use these laws to distribute the not operator over boolean expressions in Python. For example:

```
if not (age > 0 and age <= 120):
    print("Invalid age")

# can be rewritten as

if age <= 0 or age > 120:
    print("Invalid age")
```

Instead of negating each operator, we used its opposite, eliminating not altogether.

Exercise 2

1. For what values of input will this program print "True"?
2. if not input > 5:

```
3.      print("True")
```

4. For what values of absentee_rate and overall_mark will this program print "You have passed the course."?

```
5. if absentee_rate <= 5 and overall_mark >= 50:
6.      print("You have passed the course.")
```

7. For what values of x will this program print "True"?

```
8. if x > 1 or x <= 8:
9.      print("True")
```

10. Eliminate not from each of these boolean expressions:

11. not total <= 2
12. not count > 40
13. not (value > 20.0 and total != 100.0)
14. not (angle > 180 and width == 5)
15. not (count == 5 and not (value != 10) or count > 50)
16. not (value > 200 or value < 0 and not total == 0)

The None value

We often initialise a number to zero or a string to an empty string before we give it a more meaningful value. Zero and various "empty" values evaluate to False in boolean expressions, so we can check whether a variable has a meaningful value like this:

```
if (my_variable):
        print(my_variable)
```

Sometimes, however, a zero or an empty string *is* a meaningful value. How can we indicate that a variable isn't set to anything if we *can't* use zero or an empty string? We can set it to None instead.

In Python, None is a special value which means "nothing". Its type is called NoneType, and only one None value exists at a time – all the None values we use are actually the same object:

```
print(None is None) # True
```

None evaluates to False in boolean expressions. If we don't care whether our variable is None or some other value which is also false, we can just check its value like this:

```
if my_string:
    print("My string is '%s'." % my_string)
```

If, however, we want to distinguish between the case when our variable is None and when it is empty (or zero, or some other false value) we need to be more specific:

```
if my_number is not None:
    print(my_number) # could still be zero

if my_string is None:
    print("I haven't got a string at all!")
elif not my_string: # another false value, i.e. an empty string
    print("My string is empty!")
else:
    print("My string is '%s'." % my_string)
```

Switch statements and dictionary-based dispatch

if ladders can get unwieldy if they become very long. Many languages have a control statement called a switch, which tests the value of a single variable and makes a decision on the basis of that value. It is similar to an if ladder, but can be a little more readable, and is often optimised to be faster.

Python does not have a switch statement, but we can achieve something similar by using a dictionary. This example will be clearer when we have read more about dictionaries, but all we need to know for now is that a dictionary is a store of key and value pairs – we retrieve a value by its key, the way we would retrieve a list element by its index. Here is how we can rewrite the course code example:

```
DEPARTMENT_NAMES = {
    "CSC": "Computer Science",
    "MAM": "Mathematics and Applied Mathematics",
    "STA": "Statistical Sciences", # Trailing commas like this are allowed in Python!
}

if course_code in DEPARTMENT_NAMES: # this tests whether the variable is one of the dictionary's keys
    print("Department: %s" % DEPARTMENT_NAMES[course_code])
else:
    print("Unknown course code: %s" % course_code)
```

We are not limited to storing simple values like strings in the dictionary. In Python, functions can be stored in variables just like any other object, so we can even use this dispatch method to execute completely different statements in response to different values:

```
def reverse(string):
    print("'%s' reversed is '%s'." % (string, string[::-1]))

def capitalise(string):
    print("'%s' capitalised is '%s'." % (string, string.upper()))

ACTIONS = {
    "r": reverse, # use the function name without brackets to refer to the function without calling it
    "c": capitalise,
}

my_function = ACTIONS[my_action] # now we retrieve the function
my_function(my_string) # and now we call it
```

The conditional operator

Python has another way to write a selection in a program – the conditional operator. It can be used within an expression (i.e. it can be evaluated) – in contrast to if and if-else, which are just statements and not expressions. It is often called the *ternary* operator because it has *three* operands (binary operators have two, and unary operators have one). The syntax is as follows:

true expression if condition else false expression

For example:

```
result = "Pass" if (score >= 50) else "Fail"
```

This means that if score is at least 50, result is assigned "Pass", otherwise it is assigned "Fail". This is equivalent to the following if statement:

```
if (score >= 50):
        result = “Pass”
    else:
        result = “Fail”
```

The ternary operator can make simple if statements shorter and more legible, but some people may find this code harder to understand. There is no functional or efficiency difference between a normal if-else and the ternary operator. You should use the operator sparingly.

Exercise 3

1. Rewrite the following fragment as an if-ladder (using elif statements):

```
2. if temperature < 0:
3.     print("Below freezing")
4. else:
5.     if temperature < 10:
6.         print("Very cold")
7.     else:
8.         if temperature < 20:
9.             print(Chilly)
10.             else:
11.                 if temperature < 30:
12.                     print("Warm")
13.                 else:
14.                     if temperature < 40:
15.                         print("Hot")
16.                     else:
17.                         print("Too hot")
```

18. Write a Python program to assign grades to students at the end of the year. The program must do the following:

    1. Ask for a student number.
    2. Ask for the student's tutorial mark.
    3. Ask for the student's test mark.
    4. Calculate whether the student's average so far is high enough for the student to be permitted to write the examination. If the average (mean) of the tutorial and test marks is lower than 40%, the student should automatically get an F grade, and the program should print the grade and exit without performing the following steps.
    5. Ask for the student's examination mark.
    6. Calculate the student's final mark. The tutorial and test marks should count for 25% of the final mark each, and the final examination should count for the remaining 50%.
    7. Calculate and print the student's grade, according to the following table:

Weighted final score	Final grade
80 <= mark <= 100	A
70 <= mark < 80	B

Weighted final score	Final grade
60 <= mark < 70	C
50 <= mark < 60	D
mark < 50	E

Answers to exercises

Answer to exercise 1

1.

1. `if (x > 4)` – valid
2. `if x == 2` – valid (brackets are not compulsory)
3. `if (y =< 4)` – invalid (`=<` is not a valid operator; it should be `<=`)
4. `if (y = 5)` – invalid (`=` is the assignment operator, not a comparison operator)
5. `if (3 <= a)` – valid
6. `if (1 - 1)` – valid (`1 - 1` evaluates to zero, which is false)
7. `if ((1 - 1) <= 0)` – valid
8. `if (name == "James")` – valid

2. The program will print out:

```
is greater

than 3.
```

This happens because the last two print statements are not indented – they are outside the `if`statement, which means that they will always be executed.

3.

We don't have to compare variables to boolean values and compare them to `True` explicitly. This will be done implicitly if we just evaluate the variable in the condition of the `if`statement:

```
1. if a:
2.     print("a is true")
```

We set `a` to the same value whether we execute the `if` block or the `else` block, so we can move this line outside the `if` statement and only write it once.

```
3. if x > 50:
4.     b += 1
5. else:
```

```
6.      b -= 1
7. a = 5
```

Answer to exercise 2

1. The program will print "True" if input is less than or equal to 5.
2. The program will print "You have passed the course." if absentee_rate is less than or equal to 5and overall_mark is greater than or equal to 50.
3. The program will print "True" for any value of x.
4. total > 2
5. count <= 40
6. value <= 20.0 or total == 100.0
7. angle <= 180 or width != 5
8. (count != 5 or value != 10) and count <= 50
9. value <= 200 and (value >= 0 or total == 0)

Answer to exercise 3

```
1. if temperature < 0:
2.     print("Below freezing")
3. elif temperature < 10:
4.     print("Very cold")
5. elif temperature < 20:
6.     print(Chilly)
7. elif temperature < 30:
8.     print("Warm")
9. elif temperature < 40:
10.         print("Hot")
11.     else:
12.         print("Too hot")
```

Here is an example program:

```
13. student_number = input("Please enter a student number: ")
14. tutorial_mark = float(input("Please enter the student's tutorial mark: "))
15. test_mark = float(input("Please enter the student's test mark: "))
16.
```

```
17. if (tutorial_mark + test_mark) / 2 < 40:
18.     grade = "F"
19. else:
20.     exam_mark = float(input("Please enter the student's final examination mark: "))
21.     mark = (tutorial_mark + test_mark + 2 * exam_mark) / 4

22.     if 80 <= mark <= 100:
23.         grade = "A"
24.     elif 70 <= mark < 80:
25.         grade = "B"
26.     elif 60 <= mark < 70:
27.         grade = "C"
28.     elif 50 <= mark < 60:
29.         grade = "D"
30.     else:
31.         grade = "E"

    print "%s's grade is %s." % (student_number, grade)
```