Conditional Statements

Last updated on 2024-08-05 | Edit this page

Download Chapter notebook (ipynb)

OVERVIEW

Questions

- What are conditional statements?
- How conditional statements are used to make decisions?
- Why indentation is so important in Python?
- Is there any hierarchical importance of conditional statements?

Objectives

- Understand the logic behind using conditional statements.
- Practice conditional statements.
- Learning structuring code using correct indentation.
- Understanding the hierarchy of conditional statements.

This chapter assumes that you are familiar with the following concepts in Python 3:

CHECKLIST

- ☑I/O Operations
- ☑Variables And Types
- ☑Mathematical Operation

When we construct logical expressions, we almost always do so because we need to test something. The definition of a process through which we test our logical expressions and provide directives on how to proceed is known in computer science as a *conditional statement*. Conditional statements are a feature of programming languages. This means that although their definitions and grammar may vary slightly from one programming language to another, their principles are almost universally identical.

Being a high-level programming language, defining conditional statements is very easy in Python. Before we start, however, let us briefly review the way conditional statements actually work. To help us with that, we use flowchart diagrams.

REMEMBER

The term *conditional statements* is often used in relation to imperative programming languages. In functional programming, however, it is more common to refer to them as *conditional expressions* or *conditional constructs*. Python supports both imperative and functional programming.

EXAMPLE: ALGORITHMS IN DAILY LIFE

We use algorithms in our life every day without realising it.

Suppose we enter a room poorly lit room to work. The first things that we notice is that the room is insufficiently lit. We check to see whether or not the light is on; *if* not, we find the switch to turn on the light. Likewise, *if* the light is on, we go ahead and turn on the desk lamp before we proceed with our business.

PROCESSES OF ALGORITHMS

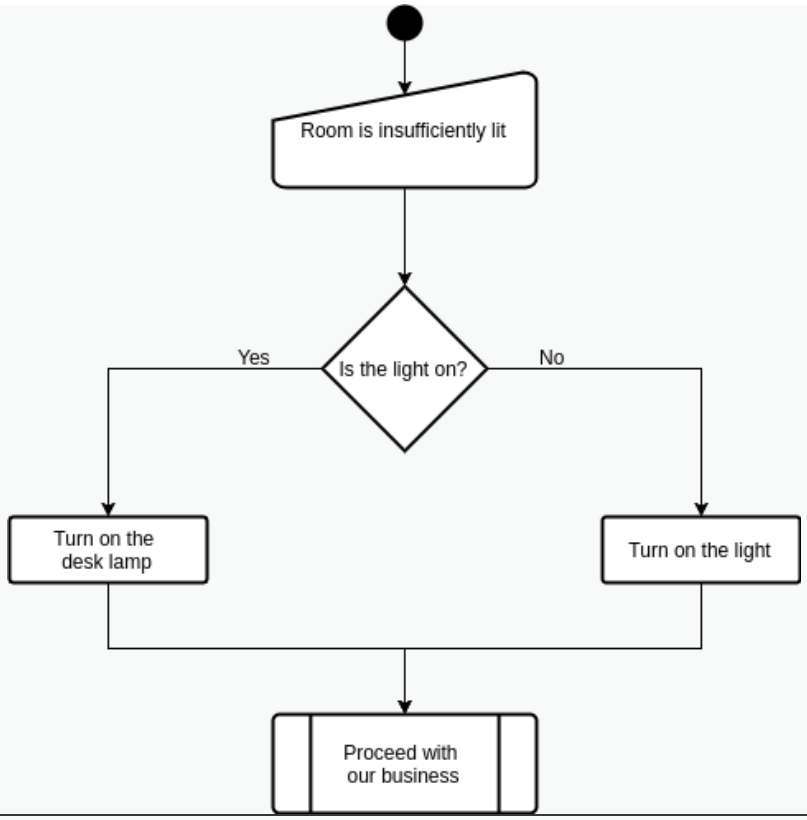
This scenario may be perceived as a set of processes. These processes include a set of procedures that may be outlined as follows:

1. `Action`: Enter the room.
2. `Input`: Determine that the room is insufficiently lit.
3. `Condition`: Is the light switched on?

- NO: `Action`: turn on the lights,
- YES: `Action`: turn on the desk lamp.

4. `Action`: Proceed with our business.

Now that we know what procedures are involved, we can draw a flowchart of the process:



Flowchart

Programming is not merely a hard skill. It is the door to a different way of thinking that enables one to break complex procedures down to simple, stepwise components and tasks. Flowcharts help us perceive processes as computers do — that is, one task or component at a time. As we progress in programming, our brains develop the ability to think in a procedural way. This is called *algorithmic thinking*, and is one of the most important soft-skills that a programmer can develop.

There are international competitions and comprehensive courses dedicated to this concept. At the end of the day, however, one can only acquire a skill through practice.

ADVANCED TOPIC

If you are keen to learn more about algorithms and algorithmic thinking, or just want to try out some of the problems, you may want to look into some of the past competition papers on Computational and Algorithmic Thinking (CAT) published by the Australian Mathematics Trust.

Exercise is provided to give you an idea of the type of problems that may be tackled in a procedural way.

PRACTICE EXERCISE 1

On a distant planet, the dominant carnivore, the zab, is nearing extinction. The number of zabs born in any year is one more than the (positive) difference between the number born in the previous year and the number born in the year before that.

Examples

- If 7 zabs were born last year and 5 the year before, 3 would be born this year.
- If 7 zabs were born last year and 10 the year before, 4 would be born this year.

If 2 zabs were born in the year 2000 and 9 zabs were born in 2001. What is the first year after 2000 when just 1 zab will be born?

a. 2009

b. 2011

c. 2013

d. 2015

e. 2017

Credit: This question is taken from the 2011 Computational and Algorithmic Thinking (CAT) Competition held by the Australian Mathematics Trust.}

Solution

To obtain the answer, we may write an algorithm in a pseudo-code format as follows:

```
  let a_total = 2

  let b_total = 9

  let current_total = absolute(a_total - b_total) + 1

  let a_total = b_total

  let b_total = current_total

  let current_year = 2002


  do {

    current_total = absolute(a_total - b_total) + 1

    a_total = b_total

    b_total = current_total

    current_year = current_year + 1

                } while current_total > 1


display current_year
```

Given:

```
  year = 2000; a_total = 2
  year = 2001; b_total= 9
```

the above process with repeat the section written in curly brackets for as long as current_total > 1:

```
  current_year: 2002; a_total = 2, b_total = 9, current_total = 8
```

Is current_total > 1 ? Yes:

```
  current_year: 2003; a_total = 9, b_total = 8; current_total = 2
```

Is current_total > 1 ? Yes:

```
  current_year: 2004; a_total = 8; b_total = 2; current_total = 7
Is current_total > 1 ? Yes:
  current_year: 2005; a_total = 2; b_total = 7; current_total = 6
Is current_total > 1 ? Yes:
  current_year: 2006; a_total = 7; b_total = 6; current_total = 2
Is current_total > 1 ? Yes:
  current_year: 2007; a_total = 6; b_total = 2; current_total = 5
Is current_total > 1 ? Yes:
  current_year: 2008; a_total = 2; b_total = 5; current_total = 4
Is current_total > 1 ? Yes:
  current_year: 2009; a_total = 5; b_total = 4; current_total = 2
Is current_total > 1 ? Yes:
  current_year: 2010; a_total = 4; b_total = 2; current_total = 3
Is current_total > 1 ? Yes:
  current_year: 2011; a_total = 2; b_total = 3; current_total = 2
Is current_total > 1 ? Yes:
  current_year: 2012; a_total = 3; b_total = 2; current_total = 2
Is current_total > 1 ? Yes:
  current_year: 2013; a_total = 2; b_total = 2; current_total = 1
Is current_total > 1 ? No:
```

The correct answer is **c)** 2013.

If this algorithm/pseudo-code is translated to Python language, it will look like this:

PYTHON

```python
a_total = 2

b_total = 9

current_year = 2002

current_total = abs(a_total - b_total) + 1
a_total = b_total
b_total = current_total

while (current_total > 1):
    current_total = abs(a_total - b_total) + 1
    a_total = b_total
    b_total = current_total
    current_year = current_year + 1

print(current_year)
```

OUTPUT

```
2013
```

REMEMBER

There is almost always more than one answer to any algorithmic problem; some answer might even be more efficient than other. The less repetition there is, the better more efficient an algorithm is considered to be.

Conditions in Python

`if` statments

To implement conditional statements in Python, we use 3 syntaxes:

- To initiate the statement, we use the syntax `if` followed by the condition and a colon;

PYTHON

```python
students_present = 15

# Conditional statement:
if students_present > 10:  # Initiation
    # Directive (must be indented).
    print('More than 10 students are present.')
```

OUTPUT

```
More than 10 students are present.
```

- To create an alternative condition after the first condition has been defined, we use the syntax `elif` followed by the new condition and a colon;

PYTHON

```python
students_present = 5

# Conditional statement:
if students_present > 10:  # Initiation

    # Directive (must be indented).
    print('More than 10 students are present.')

elif 0 < students_present < 10:
    print('Less than 10 students are present.')
```

OUTPUT

```
Less than 10 students are present.
```

- To introduce a default — i.e. where none of the above are `True`, we use the syntax `else`.

PYTHON

```python
students_present = 0

# Conditional statement:
if students_present > 10:  # Initiation
    # Directive (must be indented).
    print('More than 10 students are present.')

elif 0 < students_present < 10:  # Alternative condition
    # Alternative directive (must be indented).
    print('Less than 10 students are present.')

else:  # Default (none of the conditions are met).
    # Directive (must be indented).
    print('There is no one!')
```

OUTPUT

```
There is no one!
```

REMEMBER

We can use disjunctions or conjunctions, as discussed in topic Disjunctions and Conjunctions, to test for more than one condition at a time.

Indentation Rule

PEP-8: Indentation

Always use 4 spaces for indentation. Indentations are how the Python interpreter determines the code hierarchy. A consistent hierarchy is therefore essential for the interpreter to parse and execute our code.

The indented part of the code is known as a *block*. A block represents a part of the code that always "belongs" to (is the child process of) the first *un*indented (dedented) line that precedes it. In other words, the action(s) within a conditional statement (actions that are subject to a specific condition) must always be *indented*:

PYTHON

```python
value = 10

# Statement A:
if value > 0:

    # First dedented line before the block.
    # This is a block, and it belongs to the
    # preceding "if" (Statement A):
    print('The value is positive.')

    # We can have nested blocks too.
    # Statement B:
    if value > 9:
    # First dedented line before the block.
    # This is another block (nested).
    # This block belongs to the preceding "if" (Statement B).
        print('The value is not a single digit.')

    # Introducing a default behaviour for Statement B:
    else:
    # This block belongs to the preceding "else".
        print('The value is a single digit.')

# Introducing an alternative condition for Statement A:
elif value < 0:
# This block belongs to the preceding "elif".
    print('The value is negative.')

# Introducing a default behaviour for Statement A:
else:
# This block belongs to the preceding "else".
    print('The value is zero.')
```

OUTPUT

```
The value is positive.
The value is not a single digit.
```

It is not a good practice to have too many nested indentation. This would make the code more difficult to read. A rule of thumb is that you should not need more than 4 nested indentations in your code. If you do, you should reconsider the code structure to somehow simplify the process.

On that note, where possible, it is better to use conjunctions and disjunctions, or implement alternative conditions using `elif` instead of creating nested conditional statements. We can therefore restructure the previous example in a better, more coherent way as follows:

PYTHON

```python
value = 10

if value > 9:
  print('The value is positive.')
  print('The value is not a single digit.')
elif value > 0:
  print('The value is positive.')
  print('The value is a single digit.')
elif value < 0:
  print('The value is negative.')
else:
  print('The value is zero.')
```

OUTPUT

```
The value is positive.
The value is not a single digit.
```

It is customary and also best practice to use 4 spaces for indentation in Python. It is also paramount that all indentations throughout the code are consistent; that is, you may not use 4 spaces here and 3 spaces somewhere else in your code. Doing so will cause an `IndentationError` to be raised. It is recommended to *not* use `Tab` to indent your code; it is regarded as a bad practice in Python.

```python
value = 10

if value > 0:
    print('The value is: ')  # Indented with 4 spaces.
   print('POSITIVE.')  # Indented with 3 spaces.
```

```
File <STDIN>, line 5
    print('POSITIVE.')  # Indented with 3 spaces.
                                                  ^
IndentationError: unindent does not match any outer indentation level
```

CALLOUT

`Tab` indentations represent different number of spaces on different computers and operating systems. It is therefore more than likely that they will lead to `IndentationError`. Additionally, Python 3 disallows the mixing of **tab** and **space** indentations. Some Python IDEs such as *PyCharm* automatically convert `Tab` indentations to 4 spaces. Some other IDEs (e.g. Jupyter) typically highlight `Tab` indentations to explicitly distinguish them and thereby notify the programmer of their existence. However, more often than not, IDEs and text editors tend to *ignore* this, which amounts to inconsistencies and subsequently `IndentationError`. This is a very common difficulty that new Python programmers face, and can be very confusing if not handled correctly.

PRACTICE EXERCISE 2

In previous chapter, Practice Exercise 11, we explored the implication of `CAG` repeats in Huntington's disease. We also created a polynucleotide chain containing 36 repetition of the `CAG` codons.

Write a conditional statement that tests the length of a polyQ tract to determine the *classification* and the *disease status* based on the following Table:

Number of repeats	Classification	Disease Status
< 26	Normal	Unaffected
27 — 35	Intermediate	Unaffected
36 — 40	Reduced Penetrance	+/- Affected
> 40	Full Penetrance	Affected

Using the technique you used in Practice Exercise 11, create 5 polyQ tracts containing 26, 15, 39, 32, 36, and 54 codons. Use these polynucleotide chains to test your conditional statement.

Display the result for each chain in the following format:

```
PolyQ chain with XXX number of CAG codons:
Status: XXX
Classification: XXX
```

Hint: The length of a polyQ tract represents the number of nucleotides, not the number of `CAG` codons. See task 4 of Practice Exercise 11 for additional information.

Solution

PYTHON

```python
#Constructing the codons:

glutamine_codon = 'CAG'

polyq_codons = glutamine_codon * 26


#Determining the length of our codon:

signle_codon = len('CAG')

len_polyq = len(polyq_codons)

polyq = len_polyq / signle_codon

#Constructing the conditional statement:

NORMAL = 26
INTERMEDIATE = 35
REDUCED_PENETRANCE = 40

classification = str()
status = str()

if polyq < NORMAL:

    classification, status = 'Normal', 'Unaffected'

elif polyq <= INTERMEDIATE:

    classification, status = 'Intermediate', 'Unaffected'

elif polyq <= REDUCED_PENETRANCE:
    classification, status = 'Reduced Penetrance', '+/- Affected'

else:
    classification, status = 'Full Penetrance', 'Affected'


#Displaying the results:

print('PolyQ chain with', polyq, 'number of CAG codons:')
print('Classification:', classification)
print('Status:', status)
```

```
#Repeat this with 15, 39, 32, 36, and 54 codons.
```

OUTPUT

```
PolyQ chain with 26.0 number of CAG codons:
Classification: Intermediate
Status: Unaffected
```

Hierarchy of conditional statements

The hierarchy of conditional statement is always the same. We start the statement with an `if` syntax (initiation). This is the only essential part to implement a conditional statement. Other parts include the `elif` and the `else` syntaxes. These are the non-essential part, and we implement these as and when needed. It is, however, important that we adhere to the correct order when we implement these:

- *Always* start with the initiation syntax `if`.
- *Where needed,* implement as many alternative conditions as necessary `elif`.
- *Where needed,* implement a default behaviour using `else`}.

In an `if...elif...else` hierarchy, once one condition in the hierarchy is `True`, all *subsequent* conditions in that group are skipped and would no longer be checked.

In the following example, the first condition is `True`, therefore its corresponding block is executed and the rest of this conditional statement is skipped:

PYTHON

```
TATA_BOX = 'TATA'

promoter_region = 'GTAACTGTGGTATAATCGT'

if TATA_BOX in promoter_region:
    # This condition is "True", so this
    # and only this block is executed.
    print('There is a "TATA" box in this promoter region.')
else:
    # The last condition was "False", so this
    # block is skipped.
    print('There is no "TATA" box in this promoter region.')
```

OUTPUT

```
There is a "TATA" box in this promoter region.
```

REMEMBER

We already know from subsection Logical Operations that the value of a *boolean* (`bool`) variable is either `False` or `True`.

We have also learned that in conditional statements, we use *double* equals} or `... == ...` to test for equivalence. So naturally, one could test for the *truth value* of a `bool` variables as follows:

PYTHON

```
variable = False

if variable == False:
    print('The variable is False.')
```

OUTPUT

```
The variable is False.
```

This works, and it looks simple enough. However, this is the **wrong approach** for testing the value of `bool` variables and should *not* be used. Whilst the answer is correct in the above example, using *double equals* for testing *boolean* variables can sometimes produce incorrect results.

The correct way to test the *truth value* of a *boolean* variable is by using `is` as follows:

PYTHON

```
variable = False

if variable is False:
    print('The variable is False.')
```

OUTPUT

```
The variable is False.
```

and the negative test is:

PYTHON

```
variable = True

if variable is not False:

    print('The variable is True.')
```

OUTPUT

```
The variable is True.
```

In short; as far as *boolean* variables are concerned, we should always use `is` or `is not` to test for their *truth value* in a conditional statement.

Consequently, we can now write the example algorithm (room and light) as follows:

PYTHON

```
light_status = False

if light_status is True:

    action = 'The light is on; you may want to turn off the desk light.'

else:
    action = 'The light is off... turn it on.'

print(action)
```

OUTPUT

```
The light is off... turn it on.
```

There are a few very popular shorthands in Python that you should be familiar with when writing or reading conditional statements:

In an `if` statement, Python expects the result of the condition to be `True`. As result of that, we can simplify the above example as follows:

PYTHON

```
light_status = False

if light_status:

    action = 'The light is on; you may want to turn off the desk light.'

else:

    action = 'The light is off... turn it on.'

print(action)
```

OUTPUT

```
The light is off... turn it on.
```

Sometime, however, we might need to test for a `False` outcome. To do so, we can write a negated conditions as described in subsection Negation instead:

PYTHON

```
# Note that we have changed the order of the condition
# and added a "not" before "light_status"
if not light_status:

    action = 'The light is off... turn it on.'

else:

    action = 'The light is on; you may want to turn off the desk light.'

print(action)
```

OUTPUT

```
The light is off... turn it on.
```

EXAMPLE: A FAMILIAR SCENARIO

Suppose we want to determine the classification of the final mark for a student.

The classification protocol is as follows:

- Above 70%: First class.
- Between 60% and 70%: Second class (upper division).
- Between 50% and 60%: Second class (lower division).
- Between 40% and 50%: Pass.
- Below 40%: Fail.

PYTHON

```python
mark = 63

# Thresholds
first = 70
second_upper = 60
second_lower = 50
passed = 40  # "pass" is a syntax in Python.

if mark >= first:
    classification = 'First Class'
elif mark >= second_upper:
    classification = 'Second Class (upper division)'
elif mark >= second_lower:
    classification = 'Second Class (lower division)'
elif mark >= passed:
    classification = 'Pass'
else:
    classification = 'Fail'

print('The student obtained a', classification, 'in their studies.')
```

OUTPUT

```
The student obtained a Second Class (upper division) in their studies.
```

Exercises

END OF CHAPTER EXERCISES

1. Protein Kinases have a phosphorylation site and a consensus sequence has been determined for these sites (Rust and Thompson, 2012). All the proteins incorporate either a Serine or a Threonine residue that gets phosphorylated. Naturally, the consensus sequence for each protein varies slightly from that of other proteins.

When studying a polypeptide in the lab, a colleague realised that it has a phosphorylated Serine. So they tried to sequence the polypeptide, and managed to obtain a sequence for the protein:

```
kinase_peptide = (
        "PVWNETFVFNLKPGDVERRLSVEVWDWDRTSRNDFMGAMSFGVSELLK"
        "APVDGWYKLLNQEEGEYYNVPVADADNCSLLQKFEACNYPLELYERVR"
        "MGPSSSPIPSPSPSPTDPKRCFFGASPGRLHISDFSFLMRRRKGSFGK"
        "VMLAERRGSDELYAIKILKKDVIVQDDDVDCTLVEKRVLALGGRGPGG"
        "RPHFLTQLHSTFQTPDRLYFVMEYVTGGDLMYHIQQLGKFKEPHAAFY"
        "AAEIAIGLFFLHNQGIIYRDLKLDNVMLDAEGHIKITDFGMCKENVF"
               )
```

Desperate to find a match, and knowing that we are good at doing `computer stuff`, they asked us if we can help them identify what protein kinase does the sequence correspond to?

So we extract the consensus sequence of 3 protein kinases from the paper:

- PKC-η: either `RKGSFRR` or `RRRSFRR`
- PKC-γ: either `RRRKGSF` or `RRRKKSF`
- DMPK-E: one of `KKRRRSL`, `RKRRRSL`, `KKRRRSV`, or `RKRRRSV`.

Now all we need is to write a conditional statement in Python to identify which of the above protein kinases, if any, does our sequence correspond to. That is, which one of the consensus peptides exists in our mystery sequence?

If there is a match, our programme should display the name of the corresponding protein kinase; otherwise, it should say `No matches found` for good measures.

Solution

KEY POINTS

- Python used `if, elif and else` as conditional statements.
- In Python, there is an indentation rule of 4 spaces.
- The hierarchy of conditional statement is always the same.