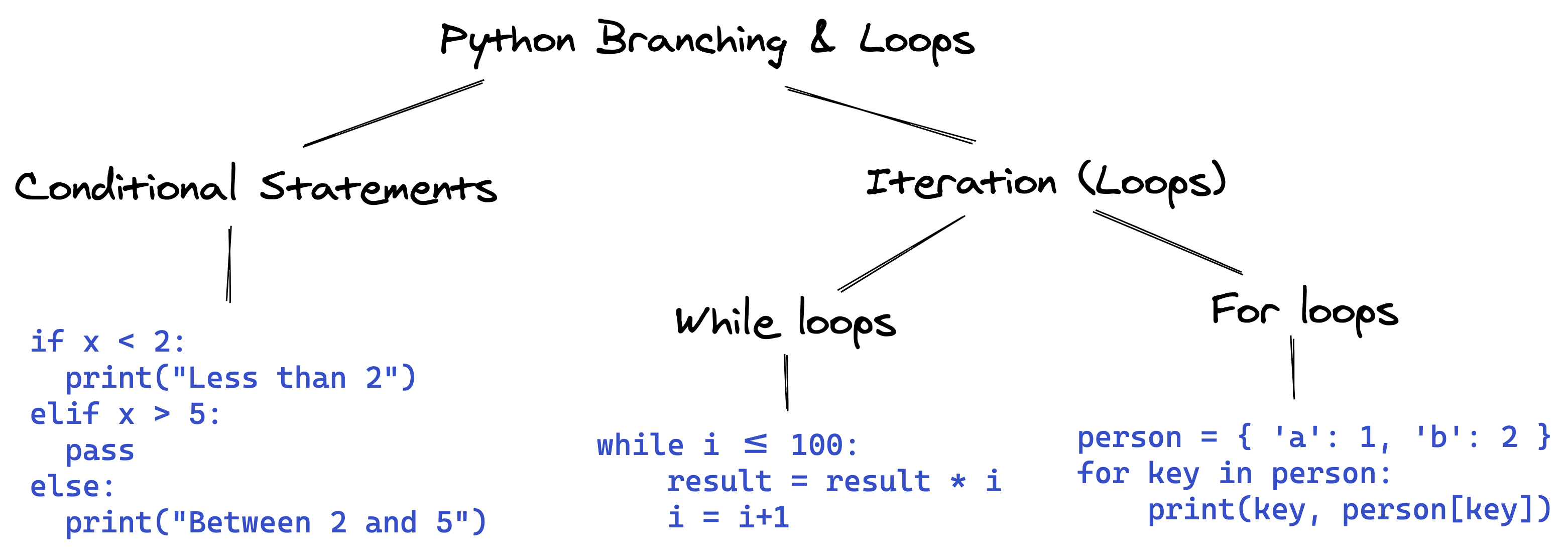
Branching using Conditional Statements and Loops in Python



Part 3 of "Data Analysis with Python: Zero to Pandas"

This tutorial series is a beginner-friendly introduction to programming and data analysis using the Python programming language. These tutorials take a practical and coding-focused approach. The best way to learn the material is to execute the code and experiment with it yourself. Check out the full series here:

1. [First Steps with Python and Jupyter](https://jovian.ai/aakashns/first-steps-with-python)
2. [A Quick Tour of Variables and Data Types](https://jovian.ai/aakashns/python-variables-and-data-types)
3. [Branching using Conditional Statements and Loops](https://jovian.ai/aakashns/python-branching-and-loops)
4. [Writing Reusable Code Using Functions](https://jovian.ai/aakashns/python-functions-and-scope)
5. [Reading from and Writing to Files](https://jovian.ai/aakashns/python-os-and-filesystem)
6. [Numerical Computing with Python and Numpy](https://jovian.ai/aakashns/python-numerical-computing-with-numpy)
7. [Analyzing Tabular Data using Pandas](https://jovian.ai/aakashns/python-pandas-data-analysis)
8. [Data Visualization using Matplotlib & Seaborn](https://jovian.ai/aakashns/python-matplotlib-data-visualization)
9. [Exploratory Data Analysis - A Case Study](https://jovian.ai/aakashns/python-eda-stackoverflow-survey)

This tutorial covers the following topics:

* Branching with if, else and elif
* Nested conditions and if expressions
* Iteration with while loops
* Iterating over containers with for loops
* Nested loops, break and continue statements

How to run the code

This tutorial is an executable [Jupyter notebook](https://jovian.ai/outlink?url=https%3A%2F%2Fjupyter.org" \t "_blank) hosted on [Jovian](https://www.jovian.ai/). You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on [Google Colab](https://jovian.ai/outlink?url=https%3A%2F%2Fcolab.research.google.com) or [Kaggle](https://jovian.ai/outlink?url=https%3A%2F%2Fkaggle.com" \t "_blank) to use these platforms.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up [Python](https://jovian.ai/outlink?url=https%3A%2F%2Fwww.python.org), download the notebook and install the required libraries. We recommend using the [Conda](https://jovian.ai/outlink?url=https%3A%2F%2Fdocs.conda.io%2Fprojects%2Fconda%2Fen%2Flatest%2Fuser-guide%2Finstall%2F" \t "_blank) distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

**Jupyter Notebooks**: This tutorial is a [Jupyter notebook](https://jovian.ai/outlink?url=https%3A%2F%2Fjupyter.org" \t "_blank) - a document made of *cells*. Each cell can contain code written in Python or explanations in plain English. You can execute code cells and view the results, e.g., numbers, messages, graphs, tables, files, etc., instantly within the notebook. Jupyter is a powerful platform for experimentation and analysis. Don't be afraid to mess around with the code & break things - you'll learn a lot by encountering and fixing errors. You can use the "Kernel > Restart & Clear Output" menu option to clear all outputs and start again from the top.

Branching with if, else and elif

One of the most powerful features of programming languages is *branching*: the ability to make decisions and execute a different set of statements based on whether one or more conditions are true.

The if statement

In Python, branching is implemented using the if statement, which is written as follows:

if condition:

statement1

statement2

The condition can be a value, variable or expression. If the condition evaluates to True, then the statements within the *if block* are executed. Notice the four spaces before statement1, statement2, etc. The spaces inform Python that these statements are associated with the if statement above. This technique of structuring code by adding spaces is called *indentation*.

**Indentation**: Python relies heavily on *indentation* (white space before a statement) to define code structure. This makes Python code easy to read and understand. You can run into problems if you don't use indentation properly. Indent your code by placing the cursor at the start of the line and pressing the Tab key once to add 4 spaces. Pressing Tab again will indent the code further by 4 more spaces, and press Shift+Tab will reduce the indentation by 4 spaces.

For example, let's write some code to check and print a message if a given number is even.

In [1]:

a\_number = 34

In [2]:

if a\_number % 2 == 0:

print("We're inside an if block")

print('The given number {} is even.'.format(a\_number))

We're inside an if block The given number 34 is even.

We use the modulus operator % to calculate the remainder from the division of a\_number by 2. Then, we use the comparison operator == check if the remainder is 0, which tells us whether the number is even, i.e., divisible by 2.

Since 34 is divisible by 2, the expression a\_number % 2 == 0 evaluates to True, so the print statement under the if statement is executed. Also, note that we are using the string format method to include the number within the message.

Let's try the above again with an odd number.

In [3]:

another\_number = 33

In [4]:

if another\_number % 2 == 0:

print('The given number {} is even.'.format(a\_number))

As expected, since the condition another\_number % 2 == 0 evaluates to False, no message is printed.

The else statement

We may want to print a different message if the number is not even in the above example. This can be done by adding the else statement. It is written as follows:

if condition:

statement1

statement2

else:

statement4

statement5

If condition evaluates to True, the statements in the if block are executed. If it evaluates to False, the statements in the else block are executed.

In [5]:

a\_number = 34

In [6]:

if a\_number % 2 == 0:

print('The given number {} is even.'.format(a\_number))

else:

print('The given number {} is odd.'.format(a\_number))

The given number 34 is even.

In [7]:

another\_number = 33

In [8]:

if another\_number % 2 == 0:

print('The given number {} is even.'.format(another\_number))

else:

print('The given number {} is odd.'.format(another\_number))

The given number 33 is odd.

Here's another example, which uses the in operator to check membership within a tuple.

In [9]:

the\_3\_musketeers = ('Athos', 'Porthos', 'Aramis')

In [10]:

a\_candidate = "D'Artagnan"

In [11]:

if a\_candidate in the\_3\_musketeers:

print("{} is a musketeer".format(a\_candidate))

else:

print("{} is not a musketeer".format(a\_candidate))

D'Artagnan is not a musketeer

The elif statement

Python also provides an elif statement (short for "else if") to chain a series of conditional blocks. The conditions are evaluated one by one. For the first condition that evaluates to True, the block of statements below it is executed. The remaining conditions and statements are not evaluated. So, in an if, elif, elif... chain, at most one block of statements is executed, the one corresponding to the first condition that evaluates to True.

In [12]:

today = 'Wednesday'

In [13]:

if today == 'Sunday':

print("Today is the day of the sun.")

elif today == 'Monday':

print("Today is the day of the moon.")

elif today == 'Tuesday':

print("Today is the day of Tyr, the god of war.")

elif today == 'Wednesday':

print("Today is the day of Odin, the supreme diety.")

elif today == 'Thursday':

print("Today is the day of Thor, the god of thunder.")

elif today == 'Friday':

print("Today is the day of Frigga, the goddess of beauty.")

elif today == 'Saturday':

print("Today is the day of Saturn, the god of fun and feasting.")

Today is the day of Odin, the supreme diety.

In the above example, the first 3 conditions evaluate to False, so none of the first 3 messages are printed. The fourth condition evaluates to True, so the corresponding message is printed. The remaining conditions are skipped. Try changing the value of today above and re-executing the cells to print all the different messages.

To verify that the remaining conditions are skipped, let us try another example.

In [14]:

a\_number = 15

In [15]:

if a\_number % 2 == 0:

print('{} is divisible by 2'.format(a\_number))

elif a\_number % 3 == 0:

print('{} is divisible by 3'.format(a\_number))

elif a\_number % 5 == 0:

print('{} is divisible by 5'.format(a\_number))

elif a\_number % 7 == 0:

print('{} is divisible by 7'.format(a\_number))

15 is divisible by 3

Note that the message 15 is divisible by 5 is not printed because the condition a\_number % 5 == 0 isn't evaluated, since the previous condition a\_number % 3 == 0 evaluates to True. This is the key difference between using a chain of if, elif, elif... statements vs. a chain of if statements, where each condition is evaluated independently.

In [16]:

if a\_number % 2 == 0:

print('{} is divisible by 2'.format(a\_number))

if a\_number % 3 == 0:

print('{} is divisible by 3'.format(a\_number))

if a\_number % 5 == 0:

print('{} is divisible by 5'.format(a\_number))

if a\_number % 7 == 0:

print('{} is divisible by 7'.format(a\_number))

15 is divisible by 3 15 is divisible by 5

Using if, elif, and else together

You can also include an else statement at the end of a chain of if, elif... statements. This code within the else block is evaluated when none of the conditions hold true.

In [17]:

a\_number = 49

In [18]:

if a\_number % 2 == 0:

print('{} is divisible by 2'.format(a\_number))

elif a\_number % 3 == 0:

print('{} is divisible by 3'.format(a\_number))

elif a\_number % 5 == 0:

print('{} is divisible by 5'.format(a\_number))

else:

print('All checks failed!')

print('{} is not divisible by 2, 3 or 5'.format(a\_number))

All checks failed! 49 is not divisible by 2, 3 or 5

Conditions can also be combined using the logical operators and, or and not. Logical operators are explained in detail in the [first tutorial](https://jovian.ai/outlink?url=https%3A%2F%2Fjovian.ml%2Faakashns%2Ffirst-steps-with-python%2Fv%2F4%23C49).

In [19]:

a\_number = 12

In [20]:

if a\_number % 3 == 0 and a\_number % 5 == 0:

print("The number {} is divisible by 3 and 5".format(a\_number))

elif not a\_number % 5 == 0:

print("The number {} is not divisible by 5".format(a\_number))

The number 12 is not divisible by 5

Non-Boolean Conditions

Note that conditions do not necessarily have to be booleans. In fact, a condition can be any value. The value is converted into a boolean automatically using the bool operator. This means that falsy values like 0, '', {}, [], etc. evaluate to False and all other values evaluate to True.

In [21]:

if '':

print('The condition evaluted to True')

else:

print('The condition evaluted to False')

The condition evaluted to False

In [22]:

if 'Hello':

print('The condition evaluted to True')

else:

print('The condition evaluted to False')

The condition evaluted to True

In [23]:

if { 'a': 34 }:

print('The condition evaluted to True')

else:

print('The condition evaluted to False')

The condition evaluted to True

In [24]:

if None:

print('The condition evaluted to True')

else:

print('The condition evaluted to False')

The condition evaluted to False

Nested conditional statements

The code inside an if block can also include an if statement inside it. This pattern is called nesting and is used to check for another condition after a particular condition holds true.

In [25]:

a\_number = 15

In [26]:

if a\_number % 2 == 0:

print("{} is even".format(a\_number))

if a\_number % 3 == 0:

print("{} is also divisible by 3".format(a\_number))

else:

print("{} is not divisibule by 3".format(a\_number))

else:

print("{} is odd".format(a\_number))

if a\_number % 5 == 0:

print("{} is also divisible by 5".format(a\_number))

else:

print("{} is not divisibule by 5".format(a\_number))

15 is odd 15 is also divisible by 5

Notice how the print statements are indented by 8 spaces to indicate that they are part of the inner if/else blocks.

Nested if, else statements are often confusing to read and prone to human error. It's good to avoid nesting whenever possible, or limit the nesting to 1 or 2 levels.

Shorthand if conditional expression

A frequent use case of the if statement involves testing a condition and setting a variable's value based on the condition.

In [27]:

a\_number = 13

if a\_number % 2 == 0:

parity = 'even'

else:

parity = 'odd'

print('The number {} is {}.'.format(a\_number, parity))

The number 13 is odd.

Python provides a shorter syntax, which allows writing such conditions in a single line of code. It is known as a *conditional expression*, sometimes also referred to as a *ternary operator*. It has the following syntax:

x = true\_value if condition else false\_value

It has the same behavior as the following if-else block:

if condition:

x = true\_value

else:

x = false\_value

Let's try it out for the example above.

In [28]:

parity = 'even' if a\_number % 2 == 0 else 'odd'

In [29]:

print('The number {} is {}.'.format(a\_number, parity))

The number 13 is odd.

Statements and Expressions

The conditional expression highlights an essential distinction between *statements* and *expressions* in Python.

**Statements**: A statement is an instruction that can be executed. Every line of code we have written so far is a statement e.g. assigning a variable, calling a function, conditional statements using if, else, and elif, loops using for and while etc.

**Expressions**: An expression is some code that evaluates to a value. Examples include values of different data types, arithmetic expressions, conditions, variables, function calls, conditional expressions, etc.

Most expressions can be executed as statements, but not all statements are expressions. For example, the regular if statement is not an expression since it does not evaluate to a value. It merely performs some branching in the code. Similarly, loops and function definitions are not expressions (we'll learn more about these in later sections).

As a rule of thumb, an expression is anything that can appear on the right side of the assignment operator =. You can use this as a test for checking whether something is an expression or not. You'll get a syntax error if you try to assign something that is not an expression.

In [30]:

*# if statement*

result = if a\_number % 2 == 0:

'even'

else:

'odd'

File "<ipython-input-30-f24978c5423e>", line 2 result = if a\_number % 2 == 0: ^ SyntaxError: invalid syntax

In [31]:

*# if expression*

result = 'even' if a\_number % 2 == 0 else 'odd'

The pass statement

if statements cannot be empty, there must be at least one statement in every if and elif block. You can use the pass statement to do nothing and avoid getting an error.

In [32]:

a\_number = 9

In [33]:

if a\_number % 2 == 0:

elif a\_number % 3 == 0:

print('{} is divisible by 3 but not divisible by 2')

File "<ipython-input-33-77268dd66617>", line 2 elif a\_number % 3 == 0: ^ IndentationError: expected an indented block

In [34]:

if a\_number % 2 == 0:

pass

elif a\_number % 3 == 0:

print('{} is divisible by 3 but not divisible by 2'.format(a\_number))

9 is divisible by 3 but not divisible by 2

Save and upload your notebook

Whether you're running this Jupyter notebook online or on your computer, it's essential to save your work from time to time. You can continue working on a saved notebook later or share it with friends and colleagues to let them execute your code. [Jovian](https://www.jovian.ai/) offers an easy way of saving and sharing your Jupyter notebooks online.

In [35]:

!pip install jovian --upgrade --quiet

In [36]:

import jovian

In [37]:

jovian.commit(project='python-branching-and-loops')

[jovian] Attempting to save notebook.. [jovian] Updating notebook "aakashns/python-branching-and-loops" on <https://jovian.ai/> [jovian] Uploading notebook.. [jovian] Capturing environment.. [jovian] Committed successfully! <https://jovian.ai/aakashns/python-branching-and-loops>

Out[37]:

'<https://jovian.ai/aakashns/python-branching-and-loops>'

The first time you run jovian.commit, you may be asked to provide an API Key to securely upload the notebook to your Jovian account. You can get the API key from your [Jovian profile page](https://jovian.ai/) after logging in / signing up.

jovian.commit uploads the notebook to your Jovian account, captures the Python environment, and creates a shareable link for your notebook, as shown above. You can use this link to share your work and let anyone (including you) run your notebooks and reproduce your work.

Iteration with while loops

Another powerful feature of programming languages, closely related to branching, is running one or more statements multiple times. This feature is often referred to as *iteration* on *looping*, and there are two ways to do this in Python: using while loops and for loops.

while loops have the following syntax:

while condition:

statement(s)

Statements in the code block under while are executed repeatedly as long as the condition evaluates to True. Generally, one of the statements under while makes some change to a variable that causes the condition to evaluate to False after a certain number of iterations.

Let's try to calculate the factorial of 100 using a while loop. The factorial of a number n is the product (multiplication) of all the numbers from 1 to n, i.e., 1\*2\*3\*...\*(n-2)\*(n-1)\*n.

In [38]:

result = 1

i = 1

while i <= 100:

result = result \* i

i = i+1

print('The factorial of 100 is: {}'.format(result))

The factorial of 100 is: 93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000

Here's how the above code works:

* We initialize two variables, result and, i. result will contain the final outcome. And i is used to keep track of the next number to be multiplied with result. Both are initialized to 1 (can you explain why?)
* The condition i <= 100 holds true (since i is initially 1), so the while block is executed.
* The result is updated to result \* i, i is increased by 1 and it now has the value 2.
* At this point, the condition i <= 100 is evaluated again. Since it continues to hold true, result is again updated to result \* i, and i is increased to 3.
* This process is repeated till the condition becomes false, which happens when i holds the value 101. Once the condition evaluates to False, the execution of the loop ends, and the print statement below it is executed.

Can you see why result contains the value of the factorial of 100 at the end? If not, try adding print statements inside the while block to print result and i in each iteration.

Iteration is a powerful technique because it gives computers a massive advantage over human beings in performing thousands or even millions of repetitive operations really fast. With just 4-5 lines of code, we were able to multiply 100 numbers almost instantly. The same code can be used to multiply a thousand numbers (just change the condition to i <= 1000) in a few seconds.

You can check how long a cell takes to execute by adding the *magic* command %%time at the top of a cell. Try checking how long it takes to compute the factorial of 100, 1000, 10000, 100000, etc.

In [39]:

%%time

result = 1

i = 1

while i <= 1000:

result \*= i *# same as result = result \* i*

i += 1 *# same as i = i+1*

print(result)

402387260077093773543702433923003985719374864210714632543799910429938512398629020592044208486969404800479988610197196058631666872994808558901323829669944590997424504087073759918823627727188732519779505950995276120874975462497043601418278094646496291056393887437886487337119181045825783647849977012476632889835955735432513185323958463075557409114262417474349347553428646576611667797396668820291207379143853719588249808126867838374559731746136085379534524221586593201928090878297308431392844403281231558611036976801357304216168747609675871348312025478589320767169132448426236131412508780208000261683151027341827977704784635868170164365024153691398281264810213092761244896359928705114964975419909342221566832572080821333186116811553615836546984046708975602900950537616475847728421889679646244945160765353408198901385442487984959953319101723355556602139450399736280750137837615307127761926849034352625200015888535147331611702103968175921510907788019393178114194545257223865541461062892187960223838971476088506276862967146674697562911234082439208160153780889893964518263243671616762179168909779911903754031274622289988005195444414282012187361745992642956581746628302955570299024324153181617210465832036786906117260158783520751516284225540265170483304226143974286933061690897968482590125458327168226458066526769958652682272807075781391858178889652208164348344825993266043367660176999612831860788386150279465955131156552036093988180612138558600301435694527224206344631797460594682573103790084024432438465657245014402821885252470935190620929023136493273497565513958720559654228749774011413346962715422845862377387538230483865688976461927383814900140767310446640259899490222221765904339901886018566526485061799702356193897017860040811889729918311021171229845901641921068884387121855646124960798722908519296819372388642614839657382291123125024186649353143970137428531926649875337218940694281434118520158014123344828015051399694290153483077644569099073152433278288269864602789864321139083506217095002597389863554277196742822248757586765752344220207573630569498825087968928162753848863396909959826280956121450994871701244516461260379029309120889086942028510640182154399457156805941872748998094254742173582401063677404595741785160829230135358081840096996372524230560855903700624271243416909004153690105933983835777939410970027753472000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000 CPU times: user 905 µs, sys: 362 µs, total: 1.27 ms Wall time: 974 µs

Here's another example that uses two while loops to create an interesting pattern.

In [40]:

line = '\*'

max\_length = 10

while len(line) <= max\_length:

print(line)

line += "\*"

while len(line) > 0:

print(line)

line = line[:-1]

\* \*\* \*\*\* \*\*\*\* \*\*\*\*\* \*\*\*\*\*\* \*\*\*\*\*\*\* \*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\* \*\*\*\*\*\*\* \*\*\*\*\*\* \*\*\*\*\* \*\*\*\* \*\*\* \*\* \*

Can you see how the above example works? As an exercise, try printing the following pattern using a while loop (Hint: use string concatenation):

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*

\*\*\*

\*\*

\*

Here's another one, putting the two together:

\*

\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*

\*\*\*

\*

Infinite Loops

Suppose the condition in a while loop always holds true. In that case, Python repeatedly executes the code within the loop forever, and the execution of the code never completes. This situation is called an infinite loop. It generally indicates that you've made a mistake in your code. For example, you may have provided the wrong condition or forgotten to update a variable within the loop, eventually falsifying the condition.

If your code is *stuck* in an infinite loop during execution, just press the "Stop" button on the toolbar (next to "Run") or select "Kernel > Interrupt" from the menu bar. This will *interrupt* the execution of the code. The following two cells both lead to infinite loops and need to be interrupted.

In [41]:

*# INFINITE LOOP - INTERRUPT THIS CELL*

result = 1

i = 1

while i <= 100:

result = result \* i

*# forgot to increment i*

--------------------------------------------------------------------------- KeyboardInterrupt Traceback (most recent call last) <ipython-input-41-5234d8c241fc> in <module> 5 6 while i <= 100: ----> 7 result = result \* i 8 # forgot to increment i KeyboardInterrupt:

In [42]:

*# INFINITE LOOP - INTERRUPT THIS CELL*

result = 1

i = 1

while i > 0 : *# wrong condition*

result \*= i

i += 1

--------------------------------------------------------------------------- KeyboardInterrupt Traceback (most recent call last) <ipython-input-42-c4abf72fce4d> in <module> 5 6 while i > 0 : # wrong condition ----> 7 result \*= i 8 i += 1 KeyboardInterrupt:

break and continue statements

You can use the break statement within the loop's body to immediately stop the execution and *break* out of the loop (even if the condition provided to while still holds true).

In [43]:

i = 1

result = 1

while i <= 100:

result \*= i

if i == 42:

print('Magic number 42 reached! Stopping execution..')

break

i += 1

print('i:', i)

print('result:', result)

Magic number 42 reached! Stopping execution.. i: 42 result: 1405006117752879898543142606244511569936384000000000

As you can see above, the value of i at the end of execution is 42. This example also shows how you can use an if statement within a while loop.

Sometimes you may not want to end the loop entirely, but simply skip the remaining statements in the loop and *continue* to the next loop. You can do this using the continue statement.

In [44]:

i = 1

result = 1

while i < 20:

i += 1

if i % 2 == 0:

print('Skipping {}'.format(i))

continue

print('Multiplying with {}'.format(i))

result = result \* i

print('i:', i)

print('result:', result)

Skipping 2 Multiplying with 3 Skipping 4 Multiplying with 5 Skipping 6 Multiplying with 7 Skipping 8 Multiplying with 9 Skipping 10 Multiplying with 11 Skipping 12 Multiplying with 13 Skipping 14 Multiplying with 15 Skipping 16 Multiplying with 17 Skipping 18 Multiplying with 19 Skipping 20 i: 20 result: 654729075

In the example above, the statement result = result \* i inside the loop is skipped when i is even, as indicated by the messages printed during execution.

**Logging**: The process of adding print statements at different points in the code (often within loops and conditional statements) for inspecting the values of variables at various stages of execution is called logging. As our programs get larger, they naturally become prone to human errors. Logging can help in verifying the program is working as expected. In many cases, print statements are added while writing & testing some code and are removed later.

Let us record a snapshot of our work before continuing using jovian.commit.

In [45]:

jovian.commit()

[jovian] Attempting to save notebook.. [jovian] Updating notebook "aakashns/python-branching-and-loops" on <https://jovian.ai/> [jovian] Uploading notebook.. [jovian] Capturing environment.. [jovian] Committed successfully! <https://jovian.ai/aakashns/python-branching-and-loops>

Out[45]:

'<https://jovian.ai/aakashns/python-branching-and-loops>'

Iteration with for loops

A for loop is used for iterating or looping over sequences, i.e., lists, tuples, dictionaries, strings, and *ranges*. For loops have the following syntax:

for value in sequence:

statement(s)

The statements within the loop are executed once for each element in sequence. Here's an example that prints all the element of a list.

In [46]:

days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']

for day in days:

print(day)

Monday Tuesday Wednesday Thursday Friday

Let's try using for loops with some other data types.

In [47]:

*# Looping over a string*

for char in 'Monday':

print(char)

M o n d a y

In [48]:

*# Looping over a tuple*

for fruit in ['Apple', 'Banana', 'Guava']:

print("Here's a fruit:", fruit)

Here's a fruit: Apple Here's a fruit: Banana Here's a fruit: Guava

In [49]:

*# Looping over a dictionary*

person = {

'name': 'John Doe',

'sex': 'Male',

'age': 32,

'married': True

}

for key in person:

print("Key:", key, ",", "Value:", person[key])

Key: name , Value: John Doe Key: sex , Value: Male Key: age , Value: 32 Key: married , Value: True

Note that while using a dictionary with a for loop, the iteration happens over the dictionary's keys. The key can be used within the loop to access the value. You can also iterate directly over the values using the .values method or over key-value pairs using the .items method.

In [50]:

for value in person.values():

print(value)

John Doe Male 32 True

In [51]:

for key\_value\_pair in person.items():

print(key\_value\_pair)

('name', 'John Doe') ('sex', 'Male') ('age', 32) ('married', True)

Since a key-value pair is a tuple, we can also extract the key & value into separate variables.

In [52]:

for key, value in person.items():

print("Key:", key, ",", "Value:", value)

Key: name , Value: John Doe Key: sex , Value: Male Key: age , Value: 32 Key: married , Value: True

Iterating using range and enumerate

The range function is used to create a sequence of numbers that can be iterated over using a for loop. It can be used in 3 ways:

* range(n) - Creates a sequence of numbers from 0 to n-1
* range(a, b) - Creates a sequence of numbers from a to b-1
* range(a, b, step) - Creates a sequence of numbers from a to b-1 with increments of step

Let's try it out.

In [53]:

for i in range(7):

print(i)

0 1 2 3 4 5 6

In [54]:

for i in range(3, 10):

print(i)

3 4 5 6 7 8 9

In [55]:

for i in range(3, 14, 4):

print(i)

3 7 11

Ranges are used for iterating over lists when you need to track the index of elements while iterating.

In [56]:

a\_list = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']

for i in range(len(a\_list)):

print('The value at position {} is {}.'.format(i, a\_list[i]))

The value at position 0 is Monday. The value at position 1 is Tuesday. The value at position 2 is Wednesday. The value at position 3 is Thursday. The value at position 4 is Friday.

Another way to achieve the same result is by using the enumerate function with a\_list as an input, which returns a tuple containing the index and the corresponding element.

In [57]:

for i, val in enumerate(a\_list):

print('The value at position {} is {}.'.format(i, val))

The value at position 0 is Monday. The value at position 1 is Tuesday. The value at position 2 is Wednesday. The value at position 3 is Thursday. The value at position 4 is Friday.

break, continue and pass statements

Similar to while loops, for loops also support the break and continue statements. break is used for breaking out of the loop and continue is used for skipping ahead to the next iteration.

In [58]:

weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']

In [59]:

for day in weekdays:

print('Today is {}'.format(day))

if (day == 'Wednesday'):

print("I don't work beyond Wednesday!")

break

Today is Monday Today is Tuesday Today is Wednesday I don't work beyond Wednesday!

In [60]:

for day in weekdays:

if (day == 'Wednesday'):

print("I don't work on Wednesday!")

continue

print('Today is {}'.format(day))

Today is Monday Today is Tuesday I don't work on Wednesday! Today is Thursday Today is Friday

Like if statements, for loops cannot be empty, so you can use a pass statement if you don't want to execute any statements inside the loop.

In [61]:

for day in weekdays:

pass

Nested for and while loops

Similar to conditional statements, loops can be nested inside other loops. This is useful for looping lists of lists, dictionaries etc.

In [62]:

persons = [{'name': 'John', 'sex': 'Male'}, {'name': 'Jane', 'sex': 'Female'}]

for person in persons:

for key in person:

print(key, ":", person[key])

print(" ")

name : John sex : Male name : Jane sex : Female

In [63]:

days = ['Monday', 'Tuesday', 'Wednesday']

fruits = ['apple', 'banana', 'guava']

for day in days:

for fruit in fruits:

print(day, fruit)

Monday apple Monday banana Monday guava Tuesday apple Tuesday banana Tuesday guava Wednesday apple Wednesday banana Wednesday guava

With this, we conclude our discussion of branching and loops in Python.