python-variables-and-data-types

June 13, 2022

1 A Quick Tour of Variables and Data Types in Python

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.

This tutorial covers the following topics:

- Storing information using variables
- Primitive data types in Python: Integer, Float, Boolean, None and String
- Built-in data structures in Python: List, Tuple and Dictionary
- Methods and operators supported by built-in data types

1.0.1 How to run the code

This tutorial is an executable Jupyter notebook hosted on Jovian. 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 or Kaggle 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, download the notebook and install the required libraries. We recommend using the Conda 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 - 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.

1.1 Storing information using variables

Computers are useful for two purposes: storing information (also known as data) and performing operations on stored data. While working with a programming language such as Python, data is stored in variables. You can think of variables are containers for storing data. The data stored within a variable is called its value. Creating variables in Python is pretty easy, as we've already seen in the previous tutorial.

[1]:
```
my_favorite_color = "blue"
```

[2]:
```
my_favorite_color
```

[2]:
```
'blue'
```

A variable is created using an assignment statement. It begins with the variable's name, followed by the assignment operator = followed by the value to be stored within the variable. Note that the assignment operator = is different from the equality comparison operator ==.

You can also assign values to multiple variables in a single statement by separating the variable names and values with commas.

[3]:
```
color1, color2, color3 = "red", "green", "blue"
```

[4]:
```
color1
```

[4]:
```
'red'
```

[5]:
```
color2
```

[5]:
```
'green'
```

[6]:
```
color3
```

[6]:
```
'blue'
```

You can assign the same value to multiple variables by chaining multiple assignment operations within a single statement.

[7]:
```
color4 = color5 = color6 = "magenta"
```

[8]:
```
color4
```

[8]:
```
'magenta'
```

[9]:
```
color5
```

[9]:
```
'magenta'
```

[10]:
```
color6
```

[10]:
```
'magenta'
```

You can change the value stored within a variable by assigning a new value to it using another assignment statement. Be careful while reassigning variables: when you assign a new value to the variable, the old value is lost and no longer accessible.

[11]:
```
my_favorite_color = "red"
```

[12]:
```
my_favorite_color
```

[12]:
```
'red'
```

While reassigning a variable, you can also use the variable's previous value to compute the new value.

[13]:
```
counter = 10
```

[14]:
```
counter = counter + 1
```

[15]:
```
counter
```

[15]:
```
11
```

The pattern `var = var op something` (where `op` is an arithmetic operator like `+`, `-`, `*`, `/`) is very common, so Python provides a *shorthand* syntax for it.

[22]:
```
counter = 10
```

[23]:
```
# Same as `counter = counter + 4`
counter += 4
```

[24]:
```
counter
```

[24]:
```
14
```

[36]:
```
counter =- 5
"""+= is not same as \n=+ (does not work the same both ways)"""
```

[36]:
```
'"+= is not same as \n=+ (does not work the same both ways)'
```

[37]:
```
counter +=1
```

[38]:
```
counter
```

[38]:
```
-4
```

[39]:
```
counter
```

[39]:
```
-4
```

Variable names can be short (`a`, `x`, `y`, etc.) or descriptive (`my_favorite_color`, `profit_margin`, `the_3_musketeers`, etc.). However, you must follow these rules while naming Python variables:

- A variable's name must start with a letter or the underscore character _. It cannot begin with a number.
- A variable name can only contain lowercase (small) or uppercase (capital) letters, digits, or underscores (`a-z`, `A-Z`, `0-9`, and _).
- Variable names are case-sensitive, i.e., `a_variable`, `A_Variable`, and `A_VARIABLE` are all different variables.

Here are some valid variable names:

[40]:
```
a_variable = 23
is_today_Saturday = False
my_favorite_car = "Delorean"
the_3_musketeers = ["Athos", "Porthos", "Aramis"]
```

Let's try creating some variables with invalid names. Python prints a syntax error if your variable's name is invalid.

> **Syntax**: The syntax of a programming language refers to the rules that govern the structure of a valid instruction or *statement*. If a statement does not follow these rules, Python stops execution and informs you that there is a *syntax error*. You can think of syntax as the rules of grammar for a programming language.

[41]:
```
a variable = 23
```

```
  File "/tmp/ipykernel_57/605469086.py", line 1
    a variable = 23
      ^
SyntaxError: invalid syntax
```

[42]:
```
is_today_$aturday = False
```

```
  File "/tmp/ipykernel_57/3433388187.py", line 1
    is_today_$aturday = False
              ^
SyntaxError: invalid syntax
```

[43]:
```
my-favorite-car = "Delorean"
```

```
  File "/tmp/ipykernel_57/1843242419.py", line 1
    my-favorite-car = "Delorean"
    ^
```

```
SyntaxError: cannot assign to operator
```

[44]:
```
3_musketeers = ["Athos", "Porthos", "Aramis"]
```

```
  File "/tmp/ipykernel_57/3494872227.py", line 1
    3_musketeers = ["Athos", "Porthos", "Aramis"]
     ^
SyntaxError: invalid decimal literal
```

1.1.1 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 offers an easy way of saving and sharing your Jupyter notebooks online.

[45]:
```
# Install the jovian library
!pip install jovian --upgrade --quiet
```

[46]:
```
import jovian
```

[47]:
```
jovian.commit(project='python-variables-and-data-types')
```

```
<IPython.core.display.Javascript object>
```

```
[jovian] Creating a new project "shubhammeena712/python-variables-and-data-
types"
[jovian] Committed successfully! https://jovian.ai/shubhammeena712/python-
variables-and-data-types
```

[47]:
```
'https://jovian.ai/shubhammeena712/python-variables-and-data-types'
```

The first time you run `jovian.commit`, you'll 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 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.

1.2 Built-in data types in Python

Any data or information stored within a Python variable has a *type*. You can view the type of data stored within a variable using the `type` function.

[48]:
```
a_variable
```

```
[48]: 23
```

```
[49]: type(a_variable)
```

```
[49]: int
```

```
[50]: is_today_Saturday
```

```
[50]: False
```

```
[51]: type(is_today_Saturday)
```

```
[51]: bool
```

```
[52]: my_favorite_car
```

```
[52]: 'Delorean'
```

```
[53]: type(my_favorite_car)
```

```
[53]: str
```

```
[54]: the_3_musketeers
```

```
[54]: ['Athos', 'Porthos', 'Aramis']
```

```
[55]: type(the_3_musketeers)
```

```
[55]: list
```

Python has several built-in data types for storing different kinds of information in variables. Following are some commonly used data types:

1. Integer
2. Float
3. Boolean
4. None
5. String
6. List
7. Tuple
8. Dictionary

Integer, float, boolean, None, and string are *primitive data types* because they represent a single value. Other data types like list, tuple, and dictionary are often called *data structures* or *containers* because they hold multiple pieces of data together.

1.2.1 Integer

Integers represent positive or negative whole numbers, from negative infinity to infinity. Note that integers should not include decimal points. Integers have the type `int`.

[56]:
```
current_year = 2020
```

[57]:
```
current_year
```

[57]:
```
2020
```

[58]:
```
type(current_year)
```

[58]:
```
int
```

Unlike some other programming languages, integers in Python can be arbitrarily large (or small). There's no lowest or highest value for integers, and there's just one `int` type (as opposed to `short`, `int`, `long`, `long long`, `unsigned int`, etc. in C/C++/Java).

[59]:
```
a_large_negative_number = -233740383748329343342343173483 43
```

[60]:
```
a_large_negative_number
```

[60]:
```
-233740383748329343342343173483 43
```

[61]:
```
type(a_large_negative_number)
```

[61]:
```
int
```

1.2.2 Float

Floats (or floating-point numbers) are numbers with a decimal point. There are no limits on the value or the number of digits before or after the decimal point. Floating-point numbers have the type `float`.

[62]:
```
pi = 3.141592653589793238
```

[63]:
```
pi
```

[63]:
```
3.141592653589793
```

how ever float type data is shorter than interger as youcan see the value after decimal is truncated

[65]:
```
type(pi)
```

[65]:
```
float
```

Note that a whole number is treated as a float if written with a decimal point, even though the decimal portion of the number is zero.

[66]:
```
a_number = 3.0
```

[67]:
```
a_number
```

[67]:
```
3.0
```

[68]:
```
type(a_number)
```

[68]:
```
float
```

[69]:
```
another_number = 4.
```

[70]:
```
another_number
```

[70]:
```
4.0
```

[71]:
```
type(another_number)
```

[71]:
```
float
```

Floating point numbers can also be written using the scientific notation with an "e" to indicate the power of 10.

[72]:
```
one_hundredth = 1e-2
```

[73]:
```
one_hundredth
```

[73]:
```
0.01
```

[74]:
```
type(one_hundredth)
```

[74]:
```
float
```

[75]:
```
avogadro_number = 6.02214076e23
```

[76]:
```
avogadro_number
```

[76]:
```
6.02214076e+23
```

[77]:
```
type(avogadro_number)
```

[77]:
```
float
```

You can convert floats into integers and vice versa using the `float` and `int` functions. The operation of converting one type of value into another is called casting.

[78]:
```
float(current_year)
```

[78]:
```
2020.0
```

[79]:
```
float(a_large_negative_number)
```

[79]:
```
-2.3374038374832935e+31
```

[80]:
```
int(pi)
```

[80]: 3

[81]:
```
int(avogadro_number)
```

[81]: 602214075999999987023872

While performing arithmetic operations, integers are automatically converted to `floats` if any of the operands is a `float`. Also, the division operator `/` always returns a `float`, even if both operands are integers. Use the `//` operator if you want the result of the division to be an `int`.

[82]:
```
type(45 * 3.0)
```

[82]: float

[83]:
```
type(45 * 3)
```

[83]: int

[84]:
```
type(10/3)
```

[84]: float

[85]:
```
type(10/2)
```

[85]: float

[86]:
```
type(10//2)
```

[86]: int

1.2.3 Boolean

Booleans represent one of 2 values: `True` and `False`. Booleans have the type `bool`.

[87]:
```
is_today_Sunday = True
```

[88]:
```
is_today_Sunday
```

[88]: True

[89]:
```
type(is_today_Saturday)
```

[89]: bool

Booleans are generally the result of a comparison operation, e.g., ==, >=, etc.

[90]:
```
cost_of_ice_bag = 1.25
is_ice_bag_expensive = cost_of_ice_bag >= 10
```

[91]:
```
is_ice_bag_expensive
```

[91]:
```
False
```

[92]:
```
type(is_ice_bag_expensive)
```

[92]:
```
bool
```

Booleans are automatically converted to `int`s when used in arithmetic operations. `True` is converted to `1` and `False` is converted to `0`.

wow this is interesting you can use boolean data type to operate mathematical operations where true is treated as 1 and false as 0.

[95]:
```
5 + False
```

[95]:
```
5
```

[96]:
```
3. + True
```

[96]:
```
4.0
```

Any value in Python can be converted to a Boolean using the `bool` function.

Only the following values evaluate to `False` (they are often called *falsy* values):

1. The value `False` itself
2. The integer `0`
3. The float `0.0`
4. The empty value `None`
5. The empty text `""`
6. The empty list `[]`
7. The empty tuple `()`
8. The empty dictionary `{}`
9. The empty set `set()`
10. The empty range `range(0)`

Everything else evaluates to `True` (a value that evaluates to `True` is often called a *truthy* value).

[97]:
```
bool(False)
```

[97]:
```
False
```

[98]:
```
bool(0)
```

[98]:
```
False
```

[99]:
```
bool(0.0)
```

[99]:
```
False
```

[100]:
```
bool(None)
```

[100]:
```
False
```

[101]:
```
bool("")
```

[101]:
```
False
```

[102]:
```
bool([])
```

[102]:
```
False
```

[103]:
```
bool(())
```

[103]:
```
False
```

[104]:
```
bool({})
```

[104]:
```
False
```

[105]:
```
bool(set())
```

[105]:
```
False
```

[106]:
```
bool(range(0))
```

[106]:
```
False
```

[107]:
```
bool(True), bool(1), bool(2.0), bool("hello"), bool([1,2]), bool((2,3)),␣
↪bool(range(10))
```

[107]:
```
(True, True, True, True, True, True, True)
```

1.2.4 None

The None type includes a single value `None`, used to indicate the absence of a value. `None` has the type `NoneType`. It is often used to declare a variable whose value may be assigned later.

[109]:
```
nothing = None
```

[110]:
```
type(nothing)
```

[110]:
```
NoneType
```

1.2.5 String

A string is used to represent text (*a string of characters*) in Python. Strings must be surrounded using quotations (either the single quote `'` or the double quote `"`). Strings have the type `string`.

[111]:
```
today = "Saturday"
```

[113]:
```
yesterday = 'friday'
yesterday
```

[113]:
```
'friday'
```

[114]:
```
today
```

[114]:
```
'Saturday'
```

[115]:
```
type(today)
```

[115]:
```
str
```

You can use single quotes inside a string written with double quotes, and vice versa.

[116]:
```
my_favorite_movie = "One Flew over the Cuckoo's Nest"
```

but here you have to take care if the single quotes are used as part of the sentence the compiler might give the following error

[120]:
```
my_favorite_movie = 'one flew over the other's this will give error'
```

```
  File "/tmp/ipykernel_57/4103369622.py", line 1
    my_favorite_movie = 'one flew over the other's this will give error'
                                                 ^
SyntaxError: invalid syntax
```

to avoid this use character to skip the next character to itself

[121]:
```
my_favorite_movie = 'one flew over the other\'s this will give error'
```

[122]:
```
my_favorite_movie
```

[122]:
```
"one flew over the other's this will give error"
```

[123]:
```
my_favorite_pun = 'Thanks for explaining the word "many" to me, it means a lot.'
```

[124]:
```
my_favorite_pun
```

[124]:
```
'Thanks for explaining the word "many" to me, it means a lot.'
```

To use a double quote within a string written with double quotes, *escape* the inner quotes by prefixing them with the \ character.

[125]:
```
another_pun = "The first time I got a universal remote control, I thought to␣
↪myself \"This changes everything\"."
```

[126]:
```
another_pun
```

[126]:
```
'The first time I got a universal remote control, I thought to myself "This
changes everything".'
```

Strings created using single or double quotes must begin and end on the same line. To create multiline strings, use three single quotes `'''` or three double quotes `"""` to begin and end the string. Line breaks are represented using the newline character `\n`.

[127]:
```
yet_another_pun = '''Son: "Dad, can you tell me what a solar eclipse is?"
Dad: "No sun."'''
```

[128]:
```
yet_another_pun
```

[128]:
```
'Son: "Dad, can you tell me what a solar eclipse is?" \nDad: "No sun."'
```

Multiline strings are best displayed using the `print` function.

[129]:
```
print(yet_another_pun)
```

```
Son: "Dad, can you tell me what a solar eclipse is?"
Dad: "No sun."
```

[130]:
```
a_music_pun = """
Two windmills are standing in a field and one asks the other,
"What kind of music do you like?"

The other says,
"I'm a big metal fan."
"""
```

[131]:
```
print(a_music_pun)
```

```
Two windmills are standing in a field and one asks the other,
"What kind of music do you like?"

The other says,
"I'm a big metal fan."
```

You can check the length of a string using the `len` function.

[132]:
```
len(my_favorite_movie)
```

[132]:
```
46
```

Note that special characters like `\n` and escaped characters like `\"` count as a single character, even though they are written and sometimes printed as two characters. Even the enter hit on keyboard is counted as a character.

[133]:
```
multiline_string = """a
b"""
multiline_string
```

[133]:
```
'a\nb'
```

[134]:
```
len(multiline_string)
```

[134]:
```
3
```

A string can be converted into a list of characters using `list` function.

[135]:
```
list(multiline_string)
```

[135]:
```
['a', '\n', 'b']
```

Strings also support several list operations, which are discussed in the next section. We'll look at a couple of examples here.

You can access individual characters within a string using the `[]` indexing notation. Note the character indices go from `0` to `n-1`, where `n` is the length of the string.

[136]:
```
today = "Saturday"
```

[137]:
```
today[0]
```

[137]:
```
'S'
```

[138]:
```
today[3]
```

[138]:
```
'u'
```

[139]:
```
today[7]
```

[139]:
```
'y'
```

You can access a part of a string using by providing a `start:end` range instead of a single index in `[]`.

[140]:
```
today[5:8]
```

[140]:
```
'day'
```

```
so a golden point to note is that the in the above command we have mentioned print character
length of string = indexes in a sting + 1;
```

You can also check whether a string contains a some text using the `in` operator.

[141]:
```
'day' in today
```

[141]: True

[142]:
```
'Sun' in today
```

[142]: False

Two or more strings can be joined or *concatenated* using the + operator. Be careful while concatenating strings, sometimes you may need to add a space character " " between words.

[143]:
```
full_name = "Derek O'Brien"
```

[144]:
```
greeting = "Hello"
```

[145]:
```
greeting + full_name
```

[145]: "HelloDerek O'Brien"

[146]:
```
greeting + " " + full_name + "!" # additional space
```

[146]: "Hello Derek O'Brien!"

Strings in Python have many built-in *methods* that are used to manipulate them. Let's try out some common string methods.

> **Methods**: Methods are functions associated with data types and are accessed using the . notation e.g. `variable_name.method()` or `"a string".method()`. Methods are a powerful technique for associating common operations with values of specific data types.

The `.lower()`, `.upper()` and `.capitalize()` methods are used to change the case of the characters.

[147]:
```
today.lower()
```

[147]: 'saturday'

[159]:
```
today ='SATURDAY'
```

[168]:
```
today.lower()
today

# .lower() changes the string casing to lower but does not change the original␣
 ↪sting to lowe case please remember.
```

[168]: 'Saturday'

can someone explain why in above cell if I call the variable name today after today.lower() it does not change the string to lower case why do we have to separately thanks in advance !

so the answer is it return the string after changing the first letter case to lower if it is not already a lower case it does not change the original string

[175]:
```
today.lower()
```

[175]:
```
'saturday'
```

[176]:
```
today
```

[176]:
```
'Saturday'
```

[177]:
```
'SATURDAY'.lower()
```

[177]:
```
'saturday'
```

[178]:
```
today = 'Saturday'
today.lower()
today
```

[178]:
```
'Saturday'
```

[179]:
```
"saturday".upper()
```

[179]:
```
'SATURDAY'
```

[180]:
```
"monday".capitalize() # changes first character to uppercase
```

[180]:
```
'Monday'
```

The `.replace` method replaces a part of the string with another string. It takes the portion to be replaced and the replacement text as *inputs* or *arguments*.

[184]:
```
another_day = today.replace("Satur", "Wednes")
another_day
```

[184]:
```
'Wednesday'
```

[185]:
```
another_day
```

[185]:
```
'Wednesday'
```

Note that `replace` returns a new string, and the original string is not modified.

[186]:
```
today
```

[186]:
```
'Saturday'
```

The `.split` method splits a string into a list of strings at every occurrence of provided character(s).

[187]:
```
"Sun,Mon,Tue,Wed,Thu,Fri,Sat".split(",")
```

[187]:
```
['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
```

The `.strip` method removes whitespace characters from the beginning and end of a string.

[188]:
```
a_long_line = "       This is a long line with some space before, after,     and␣
 ↪some space in the middle..    "
```

[189]:
```
a_long_line_stripped = a_long_line.strip()
```

[190]:
```
a_long_line_stripped
```

[190]:
```
'This is a long line with some space before, after,     and some space in the
middle..'
```

The `.format` method combines values of other data types, e.g., integers, floats, booleans, lists, etc. with strings. You can use `format` to construct output messages for display.

[191]:
```
# Input variables
cost_of_ice_bag = 1.25
profit_margin = .2
number_of_bags = 500

# Template for output message
output_template = """If a grocery store sells ice bags at $ {} per bag, with a␣
 ↪profit margin of {} %,
then the total profit it makes by selling {} ice bags is $ {}."""

print(output_template)
```

```
If a grocery store sells ice bags at $ {} per bag, with a profit margin of {} %,
then the total profit it makes by selling {} ice bags is $ {}.
```

[192]:
```
# Inserting values into the string
total_profit = cost_of_ice_bag * profit_margin * number_of_bags
output_message = output_template.format(cost_of_ice_bag, profit_margin*100,␣
 ↪number_of_bags, total_profit)

print(output_message)
```

```
If a grocery store sells ice bags at $ 1.25 per bag, with a profit margin of
20.0 %,
then the total profit it makes by selling 500 ice bags is $ 125.0.
```

Notice how the placeholders `{}` in the `output_template` string are replaced with the arguments provided to the `.format` method.

It is also possible to use the string concatenation operator `+` to combine strings with other values. However, those values must first be converted to strings using the `str` function.

[193]:
```
"If a grocery store sells ice bags at $ " + cost_of_ice_bag + ", with a profit␣
 ↪margin of " + profit_margin
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
```

```
/tmp/ipykernel_57/1331907882.py in <module>
----> 1 "If a grocery store sells ice bags at $ " + cost_of_ice_bag + ", with a␣
 ↪profit margin of " + profit_margin

TypeError: can only concatenate str (not "float") to str
```

[194]:
```
"If a grocery store sells ice bags at $ " + str(cost_of_ice_bag) + ", with a␣
 ↪profit margin of " + str(profit_margin)
```

[194]: `'If a grocery store sells ice bags at $ 1.25, with a profit margin of 0.2'`

You can `str` to convert a value of any data type into a string.

[195]:
```
str(23)
```

[195]: `'23'`

[196]:
```
str(23.432)
```

[196]: `'23.432'`

[197]:
```
str(True)
```

[197]: `'True'`

[198]:
```
the_3_musketeers = ["Athos", "Porthos", "Aramis"]
str(the_3_musketeers)
```

[198]: `"['Athos', 'Porthos', 'Aramis']"`

Note that all string methods return new values and DO NOT change the existing string. You can find a full list of string methods here: https://www.w3schools.com/python/python_ref_string.asp. Do refer to this website for syntax references in future.

Strings also support the comparison operators `==` and `!=` for checking whether two strings are equal.

[199]:
```
first_name = "John"
```

[200]:
```
first_name == "Doe"
```

[200]: `False`

[201]:
```
first_name == "John"
```

[201]: `True`

[202]:
```
first_name != "Jane"
```

[202]:
```
True
```

We've looked at the primitive data types in Python. We're now ready to explore non-primitive data structures, also known as containers.

Before continuing, let us run `jovian.commit` once again to record another snapshot of our notebook.

[203]:
```
jovian.commit()
```

```
<IPython.core.display.Javascript object>
```

```
[jovian] Updating notebook "shubhammeena712/python-variables-and-data-types" on
https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/shubhammeena712/python-
variables-and-data-types
```

[203]:
```
'https://jovian.ai/shubhammeena712/python-variables-and-data-types'
```

Running `jovian.commit` multiple times within a notebook records new versions automatically. You will continue to have access to all the previous versions of your notebook, using the versions dropdown on the notebook's Jovian.

1.2.6 List

A list in Python is an ordered collection of values. Lists can hold values of different data types and support operations to add, remove, and change values. Lists have the type `list`.

To create a list, enclose a sequence of values within square brackets `[` and `]`, separated by commas.

[204]:
```
fruits = ['apple', 'banana', 'cherry']
```

[205]:
```
fruits
```

[205]:
```
['apple', 'banana', 'cherry']
```

[206]:
```
type(fruits)
```

[206]:
```
list
```

Let's try creating a list containing values of different data types, including another list.

[207]:
```
a_list = [23, 'hello', None, 3.14, fruits, 3 <= 5]
```

[208]:
```
a_list
```

[208]:
```
[23, 'hello', None, 3.14, ['apple', 'banana', 'cherry'], True]
```

[209]:
```
empty_list = []
```

[210]:
```
empty_list
```

[210]: []

To determine the number of values in a list, use the `len` function. You can use `len` to determine the number of values in several other data types.

[211]:
```
len(empty_list)
```

[211]: 0

[212]:
```
len(fruits)
```

[212]: 3

[213]:
```
print("Number of fruits:", len(fruits))
```

```
Number of fruits: 3
```

[214]:
```
len(a_list)
```

[214]: 6

[215]:
```
len(empty_list)
```

[215]: 0

You can access an element from the list using its *index*, e.g., `fruits[2]` returns the element at index 2 within the list `fruits`. The starting index of a list is 0.

[216]:
```
fruits[0]
```

[216]: 'apple'

[217]:
```
fruits[1]
```

[217]: 'banana'

[218]:
```
fruits[2]
```

[218]: 'cherry'

If you try to access an index equal to or higher than the length of the list, Python returns an `IndexError`.

[219]:
```
fruits[3]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/tmp/ipykernel_57/1511222973.py in <module>
----> 1 fruits[3]
```

```
IndexError: list index out of range
```

[220]:
```
fruits[4]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/tmp/ipykernel_57/242072258.py in <module>
----> 1 fruits[4]

IndexError: list index out of range
```

You can use negative indices to access elements from the end of a list, e.g., `fruits[-1]` returns the last element, `fruits[-2]` returns the second last element, and so on.

[221]:
```
fruits[-1]
```

[221]:
```
'cherry'
```

[222]:
```
fruits[-2]
```

[222]:
```
'banana'
```

[223]:
```
fruits[-3]
```

[223]:
```
'apple'
```

[224]:
```
fruits[-4]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/tmp/ipykernel_57/4281187226.py in <module>
----> 1 fruits[-4]

IndexError: list index out of range
```

You can also access a range of values from the list. The result is itself a list. Let us look at some examples.

[225]:
```
a_list = [23, 'hello', None, 3.14, fruits, 3 <= 5]
```

[226]:
```
a_list
```

[226]:
```
[23, 'hello', None, 3.14, ['apple', 'banana', 'cherry'], True]
```

[227]:
```
len(a_list)
```

[227]: 6

[228]:
```
a_list[2:5]
```

[228]:
```
[None, 3.14, ['apple', 'banana', 'cherry']]
```

Note that the range `2:5` includes the element at the start index 2 but does not include the element at the end index 5. So, the result has 3 values (index 2, 3, and 4).

Here are some experiments you should try out (use the empty cells below):

- Try setting one or both indices of the range are larger than the size of the list, e.g., `a_list[2:10]`
- Try setting the start index of the range to be larger than the end index, e.g., `a_list[12:10]`
- Try leaving out the start or end index of a range, e.g., `a_list[2:]` or `a_list[:5]`
- Try using negative indices for the range, e.g., `a_list[-2:-5]` or `a_list[-5:-2]` (can you explain the results?)

  The flexible and interactive nature of Jupyter notebooks makes them an excellent tool for learning and experimentation. If you are new to Python, you can resolve most questions as soon as they arise simply by typing the code into a cell and executing it. Let your curiosity run wild, discover what Python is capable of and what it isn't!

[231]:
```
a_list
```

[231]:
```
[23, 'hello', None, 3.14, ['apple', 'banana', 'cherry'], True]
```

[238]:
```
a_list[1:12]
```

[238]:
```
['hello', None, 3.14, ['apple', 'banana', 'cherry'], True]
```

if index given is beyond the length of string it will print the whole string

[239]:
```
a_list[:]
```

[239]:
```
[23, 'hello', None, 3.14, ['apple', 'banana', 'cherry'], True]
```

[240]:
```
a_list[12:]
```

[240]:
```
[]
```

[241]:
```
a_list[:3]
```

[241]:
```
[23, 'hello', None]
```

You can also change the value at a specific index within a list using the assignment operation.

[242]:
```
a_list[-2:]
```

[242]:
```
[['apple', 'banana', 'cherry'], True]
```

[243]:
```
a_list[-2:-5]
```

[243]:
```
[]
```

Why does this return an empty string ??????

[244]:
```
fruits
```

[244]:
```
['apple', 'banana', 'cherry']
```

[245]:
```
fruits[1] = 'blueberry'
```

[246]:
```
fruits
```

[246]:
```
['apple', 'blueberry', 'cherry']
```

A new value can be added to the end of a list using the `append` method.

[247]:
```
fruits.append('dates')
```

[248]:
```
fruits
```

[248]:
```
['apple', 'blueberry', 'cherry', 'dates']
```

A new value can also be inserted at a specific index using the `insert` method.

[249]:
```
fruits.insert(1, 'banana')
```

[250]:
```
fruits
```

[250]:
```
['apple', 'banana', 'blueberry', 'cherry', 'dates']
```

You can remove a value from a list using the `remove` method.

[251]:
```
fruits.remove('blueberry')
```

[252]:
```
fruits
```

[252]:
```
['apple', 'banana', 'cherry', 'dates']
```

What happens if a list has multiple instances of the value passed to `.remove`? Try it out.

[253]:
```
fruits.remove('apple','banana')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_57/949972962.py in <module>
----> 1 fruits.remove('apple','banana')

TypeError: list.remove() takes exactly one argument (2 given)
```

done

To remove an element from a specific index, use the `pop` method. The method also returns the removed element.

[254]:
```
fruits
```

[254]:
```
['apple', 'banana', 'cherry', 'dates']
```

[255]:
```
fruits.pop(1)
```

[255]:
```
'banana'
```

[259]:
```
fruits.pop(1,2,3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_57/3673365441.py in <module>
----> 1 fruits.pop(1,2,3)

TypeError: pop expected at most 1 argument, got 3
```

[256]:
```
fruits
```

[256]:
```
['apple', 'cherry', 'dates']
```

If no index is provided, the `pop` method removes the last element of the list.

[257]:
```
fruits.pop()
```

[257]:
```
'dates'
```

[258]:
```
fruits
```

[258]:
```
['apple', 'cherry']
```

You can test whether a list contains a value using the `in` operator.

[260]:
```
'pineapple' in fruits
```

[260]:
```
False
```

[261]:
```
'cherry' in fruits
```

[261]:
```
True
```

To combine two or more lists, use the + operator. This operation is also called *concatenation*.

[262]:
```
fruits
```

[262]:
```
['apple', 'cherry']
```

[263]:
```
more_fruits = fruits + ['pineapple', 'tomato', 'guava'] + ['dates', 'banana']
```

[264]:
```
more_fruits
```

[264]:
```
['apple', 'cherry', 'pineapple', 'tomato', 'guava', 'dates', 'banana']
```

To create a copy of a list, use the `copy` method. Modifying the copied list does not affect the original.

[265]:
```
more_fruits_copy = more_fruits.copy()
```

[266]:
```
more_fruits_copy
```

[266]:
```
['apple', 'cherry', 'pineapple', 'tomato', 'guava', 'dates', 'banana']
```

[267]:
```
# Modify the copy
more_fruits_copy.remove('pineapple')
more_fruits_copy.pop()
more_fruits_copy
```

[267]:
```
['apple', 'cherry', 'tomato', 'guava', 'dates']
```

[268]:
```
# Original list remains unchanged
more_fruits
```

[268]:
```
['apple', 'cherry', 'pineapple', 'tomato', 'guava', 'dates', 'banana']
```

Note that you cannot create a copy of a list by simply creating a new variable using the assignment operator =. The new variable will point to the same list, and any modifications performed using either variable will affect the other.

[269]:
```
more_fruits
```

[269]:
```
['apple', 'cherry', 'pineapple', 'tomato', 'guava', 'dates', 'banana']
```

[270]:
```
more_fruits_not_a_copy = more_fruits
```

[271]:
```
more_fruits_not_a_copy.remove('pineapple')
more_fruits_not_a_copy.pop()
```

[271]:
```
'banana'
```

[272]:
```
more_fruits_not_a_copy
```

[272]:
```
['apple', 'cherry', 'tomato', 'guava', 'dates']
```

[273]:
```
more_fruits
```

[273]: ['apple', 'cherry', 'tomato', 'guava', 'dates']

Just like strings, there are several in-built methods to manipulate a list. However, unlike strings, most list methods modify the original list rather than returning a new one. Check out some common list operations here: https://www.w3schools.com/python/python_ref_list.asp .

Following are some exercises you can try out with list methods (use the blank code cells below):

- Reverse the order of elements in a list
- Add the elements of one list at the end of another list
- Sort a list of strings in alphabetical order
- Sort a list of numbers in decreasing order

[275]:
```
more_fruits.reverse()
more_fruits
```

[275]: ['apple', 'cherry', 'tomato', 'guava', 'dates']

[276]:
```
fruits = ['apple', 'banana', 'cherry']

cars = ['Ford', 'BMW', 'Volvo']

fruits.extend(cars)
```

[277]:
```
fruits
```

[277]: ['apple', 'banana', 'cherry', 'Ford', 'BMW', 'Volvo']

[279]:
```
cars = ['Ford', 'BMW', 'Volvo']

cars.sort()

cars
```

[279]: ['BMW', 'Ford', 'Volvo']

[282]:
```
cars = ['Ford', 'BMW', 'Volvo']

cars.sort(reverse = True)

cars
```

[282]: ['Volvo', 'Ford', 'BMW']

how to use key argument in reverse method for string doubt !!

1.2.7 Tuple

A tuple is an ordered collection of values, similar to a list. However, it is not possible to add, remove, or modify values in a tuple. A tuple is created by enclosing values within parentheses (

and), separated by commas.

> Any data structure that cannot be modified after creation is called *immutable*. You can think of tuples as immutable lists.

Let's try some experiments with tuples.

[284]:
```
fruits = ('apple', 'cherry', 'dates')
```

[285]:
```
# check no. of elements
len(fruits)
```

[285]:
```
3
```

[286]:
```
# get an element (positive index)
fruits[0]
```

[286]:
```
'apple'
```

[287]:
```
# get an element (negative index)
fruits[-2]
```

[287]:
```
'cherry'
```

[288]:
```
# check if it contains an element
'dates' in fruits
```

[288]:
```
True
```

[289]:
```
# try to change an element
fruits[0] = 'avocado'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_57/184743337.py in <module>
      1 # try to change an element
----> 2 fruits[0] = 'avocado'

TypeError: 'tuple' object does not support item assignment
```

[290]:
```
# try to append an element
fruits.append('blueberry')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/tmp/ipykernel_57/3010634816.py in <module>
      1 # try to append an element
----> 2 fruits.append('blueberry')
```

```
AttributeError: 'tuple' object has no attribute 'append'
```

[291]:
```
# try to remove an element
fruits.remove('apple')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/tmp/ipykernel_57/3303018582.py in <module>
      1 # try to remove an element
----> 2 fruits.remove('apple')

AttributeError: 'tuple' object has no attribute 'remove'
```

You can also skip the parantheses (and) while creating a tuple. Python automatically converts comma-separated values into a tuple.

[292]:
```
the_3_musketeers = 'Athos', 'Porthos', 'Aramis'
```

[293]:
```
the_3_musketeers
```

[293]:
```
('Athos', 'Porthos', 'Aramis')
```

You can also create a tuple with just one element by typing a comma after it. Just wrapping it with parentheses (and) won't make it a tuple.

[294]:
```
single_element_tuple = 4,
```

[295]:
```
single_element_tuple
```

[295]:
```
(4,)
```

[296]:
```
another_single_element_tuple = (4,)
```

[297]:
```
another_single_element_tuple
```

[297]:
```
(4,)
```

[298]:
```
not_a_tuple = (4)
```

[299]:
```
not_a_tuple
```

[299]:
```
4
```

Tuples are often used to create multiple variables with a single statement.

[300]:
```
point = (3, 4)
```

[301]:
```
point_x, point_y = point
```

[302]:
```
point_x
```

[302]:
```
3
```

[303]:
```
point_y
```

[303]:
```
4
```

You can convert a list into a tuple using the `tuple` function, and vice versa using the `list` function

[304]:
```
tuple(['one', 'two', 'three'])
```

[304]:
```
('one', 'two', 'three')
```

[305]:
```
list(('Athos', 'Porthos', 'Aramis'))
```

[305]:
```
['Athos', 'Porthos', 'Aramis']
```

Tuples have just two built-in methods: `count` and `index`. Can you figure out what they do? While you could look for documentation and examples online, there's an easier way to check a method's documentation, using the `help` function.

[306]:
```
a_tuple = 23, "hello", False, None, 23, 37, "hello"
```

[308]:
```
help(a_tuple)
```

```
Help on tuple object:

class tuple(object)
 |  tuple(iterable=(), /)
 |
 |  Built-in immutable sequence.
 |
 |  If no argument is given, the constructor returns an empty tuple.
 |  If iterable is specified the tuple is initialized from iterable's items.
 |
 |  If the argument is a tuple, the return value is the same object.
 |
 |  Built-in subclasses:
 |      asyncgen_hooks
 |      RefResolutionErrorAttributes
 |      TypeCheckerAttributes
 |      UnraisableHookArgs
 |      ... and 1 other subclasses
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
```

```
|      Return self+value.
|
|  __contains__(self, key, /)
|      Return key in self.
|
|  __eq__(self, value, /)
|      Return self==value.
|
|  __ge__(self, value, /)
|      Return self>=value.
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __getitem__(self, key, /)
|      Return self[key].
|
|  __getnewargs__(self, /)
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __hash__(self, /)
|      Return hash(self).
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mul__(self, value, /)
|      Return self*value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __rmul__(self, value, /)
|      Return value*self.
```

```
 |
 |  count(self, value, /)
 |      Return number of occurrences of value.
 |
 |  index(self, value, start=0, stop=9223372036854775807, /)
 |      Return first index of value.
 |
 |      Raises ValueError if the value is not present.
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  __class_getitem__(...) from builtins.type
 |      See PEP 585
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
```

[309]:
```
help(a_tuple.count)
```

```
Help on built-in function count:

count(value, /) method of builtins.tuple instance
    Return number of occurrences of value.
```

Within a Jupyter notebook, you can also start a code cell with `?` and type the name of a function or method. When you execute this cell, you will see the function/method's documentation in a pop-up window.

[310]:
```
?a_tuple.index
```

Try using `count` and `index` with `a_tuple` in the code cells below.

[312]:
```
a_tuple.count('banana')
```

[312]: 0

[318]:
```
 a_tuple.index(23)
```

[318]: 0

1.2.8 Dictionary

A dictionary is an unordered collection of items. Each item stored in a dictionary has a key and value. You can use a key to retrieve the corresponding value from the dictionary. Dictionaries

have the type `dict`.

Dictionaries are often used to store many pieces of information e.g. details about a person, in a single variable. Dictionaries are created by enclosing key-value pairs within braces or curly brackets { and }.

[325]:
```
help(dict)
```

```
Help on class dict in module builtins:

class dict(object)
 |  dict() -> new empty dictionary
 |  dict(mapping) -> new dictionary initialized from a mapping object's
 |      (key, value) pairs
 |  dict(iterable) -> new dictionary initialized as if via:
 |      d = {}
 |      for k, v in iterable:
 |          d[k] = v
 |  dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |      in the keyword argument list.  For example:  dict(one=1, two=2)
 |
 |  Built-in subclasses:
 |      StgDict
 |
 |  Methods defined here:
 |
 |  __contains__(self, key, /)
 |      True if the dictionary has the specified key, else False.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
```

```
|
|  __ior__(self, value, /)
|      Return self|=value.
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __or__(self, value, /)
|      Return self|value.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __reversed__(self, /)
|      Return a reverse iterator over the dict keys.
|
|  __ror__(self, value, /)
|      Return value|self.
|
|  __setitem__(self, key, value, /)
|      Set self[key] to value.
|
|  __sizeof__(...)
|      D.__sizeof__() -> size of D in memory, in bytes
|
|  clear(...)
|      D.clear() -> None.  Remove all items from D.
|
|  copy(...)
|      D.copy() -> a shallow copy of D
|
|  get(self, key, default=None, /)
|      Return the value for key if key is in the dictionary, else default.
|
|  items(...)
|      D.items() -> a set-like object providing a view on D's items
```

```
 |
 |  keys(...)
 |      D.keys() -> a set-like object providing a view on D's keys
 |
 |  pop(...)
 |      D.pop(k[,d]) -> v, remove specified key and return the corresponding
value.
 |
 |      If key is not found, default is returned if given, otherwise KeyError is
raised
 |
 |  popitem(self, /)
 |      Remove and return a (key, value) pair as a 2-tuple.
 |
 |      Pairs are returned in LIFO (last-in, first-out) order.
 |      Raises KeyError if the dict is empty.
 |
 |  setdefault(self, key, default=None, /)
 |      Insert key with a value of default if key is not in the dictionary.
 |
 |      Return the value for key if key is in the dictionary, else default.
 |
 |  update(...)
 |      D.update([E, ]**F) -> None.  Update D from dict/iterable E and F.
 |      If E is present and has a .keys() method, then does:  for k in E: D[k] =
E[k]
 |      If E is present and lacks a .keys() method, then does:  for k, v in E:
D[k] = v
 |      In either case, this is followed by: for k in F:  D[k] = F[k]
 |
 |  values(...)
 |      D.values() -> an object providing a view on D's values
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  __class_getitem__(...) from builtins.type
 |      See PEP 585
 |
 |  fromkeys(iterable, value=None, /) from builtins.type
 |      Create a new dictionary with keys from iterable and values set to value.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
```

```
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __hash__ = None
```

[319]:
```
person1 = {
    'name': 'John Doe',
    'sex': 'Male',
    'age': 32,
    'married': True
}
```

[320]:
```
person1
```

[320]:
```
{'name': 'John Doe', 'sex': 'Male', 'age': 32, 'married': True}
```

Dictionaries can also be created using the `dict` function.

[326]:
```
person2 = dict(name='Jane Judy', sex='Female', age=28, married=False)
```

[327]:
```
person2
```

[327]:
```
{'name': 'Jane Judy', 'sex': 'Female', 'age': 28, 'married': False}
```

[328]:
```
type(person1)
```

[328]:
```
dict
```

Keys can be used to access values using square brackets `[` and `]`.

[329]:
```
person1['name']
```

[329]:
```
'John Doe'
```

[330]:
```
person1['married']
```

[330]:
```
True
```

[331]:
```
person2['name']
```

[331]:
```
'Jane Judy'
```

If a key isn't present in the dictionary, then a `KeyError` is *thrown*.

[332]:
```
person1['address']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/tmp/ipykernel_57/786906807.py in <module>
```

```
----> 1 person1['address']

KeyError: 'address'
```

You can also use the `get` method to access the value associated with a key.

[333]:
```
person2.get("name")
```

[333]:
```
'Jane Judy'
```

The `get` method also accepts a default value, returned if the key is not present in the dictionary.

[334]:
```
person2.get("address", "Unknown")
```

[334]:
```
'Unknown'
```

You can check whether a key is present in a dictionary using the `in` operator.

[335]:
```
'name' in person1
```

[335]:
```
True
```

[336]:
```
'address' in person1
```

[336]:
```
False
```

You can change the value associated with a key using the assignment operator.

[337]:
```
person2['married']
```

[337]:
```
False
```

[338]:
```
person2['married'] = True
```

[339]:
```
person2['married']
```

[339]:
```
True
```

The assignment operator can also be used to add new key-value pairs to the dictionary.

[340]:
```
person1
```

[340]:
```
{'name': 'John Doe', 'sex': 'Male', 'age': 32, 'married': True}
```

[341]:
```
person1['address'] = '1, Penny Lane'
```

[342]:
```
person1
```

[342]:
```
{'name': 'John Doe',
 'sex': 'Male',
```

```
 'age': 32,
 'married': True,
 'address': '1, Penny Lane'}
```

To remove a key and the associated value from a dictionary, use the pop method.

[343]:
```
person1.pop('address')
```

[343]:
```
'1, Penny Lane'
```

[344]:
```
person1
```

[344]:
```
{'name': 'John Doe', 'sex': 'Male', 'age': 32, 'married': True}
```

Dictionaries also provide methods to view the list of keys, values, or key-value pairs inside it.

[345]:
```
person1.keys()
```

[345]:
```
dict_keys(['name', 'sex', 'age', 'married'])
```

[346]:
```
person1.values()
```

[346]:
```
dict_values(['John Doe', 'Male', 32, True])
```

[347]:
```
person1.items()
```

[347]:
```
dict_items([('name', 'John Doe'), ('sex', 'Male'), ('age', 32), ('married',
True)])
```

[348]:
```
person1.items()[1]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_57/2295007945.py in <module>
----> 1 person1.items()[1]

TypeError: 'dict_items' object is not subscriptable
```

The results of `keys`, `values`, and `items` look like lists. However, they don't support the indexing operator `[]` for retrieving elements.

Can you figure out how to access an element at a specific index from these results? Try it below. *Hint: Use the `list` function*

[354]:
```
print(person1['name'][0:3])
```

```
Joh
```

[ ]:
```

```

Dictionaries provide many other methods. You can learn more about them here: https://www.w3schools.com/python/python_ref_dictionary.asp .

Here are some experiments you can try out with dictionaries (use the empty cells below): * What happens if you use the same key multiple times while creating a dictionary? * How can you create a copy of a dictionary (modifying the copy should not change the original)? * Can the value associated with a key itself be a dictionary? * How can you add the key-value pairs from one dictionary into another dictionary? Hint: See the `update` method. * Can the dictionary's keys be something other than a string, e.g., a number, boolean, list, etc.?

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

1.3 Further Reading

We've now completed our exploration of variables and common data types in Python. Following are some resources to learn more about data types in Python:

- Python official documentation: https://docs.python.org/3/tutorial/index.html
- Python Tutorial at W3Schools: https://www.w3schools.com/python/
- Practical Python Programming: https://dabeaz-course.github.io/practical-python/Notes/Contents.html

You are now ready to move on to the next tutorial: Branching using conditional statements and loops in Python

Let's save a snapshot of our notebook one final time using `jovian.commit`.

[ ]:
```
jovian.commit()
```

1.4 Questions for Revision

Try answering the following questions to test your understanding of the topics covered in this notebook:

1. What is a variable in Python?
2. How do you create a variable?
3. How do you check the value within a variable?
4. How do you create multiple variables in a single statement?
5. How do you create multiple variables with the same value?
6. How do you change the value of a variable?
7. How do you reassign a variable by modifying the previous value?
8. What does the statement `counter += 4` do?
9. What are the rules for naming a variable?

10. Are variable names case-sensitive? Do `a_variable`, `A_Variable`, and `A_VARIABLE` represent the same variable or different ones?
11. What is Syntax? Why is it important?
12. What happens if you execute a statement with invalid syntax?
13. How do you check the data type of a variable?
14. What are the built-in data types in Python?
15. What is a primitive data type?
16. What are the primitive data types available in Python?
17. What is a data structure or container data type?
18. What are the container types available in Python?
19. What kind of data does the Integer data type represent?
20. What are the numerical limits of the integer data type?
21. What kind of data does the float data type represent?
22. How does Python decide if a given number is a float or an integer?
23. How can you create a variable which stores a whole number, e.g., 4 but has the float data type?
24. How do you create floats representing very large (e.g., 6.023 x 10^23) or very small numbers (0.000000123)?
25. What does the expression `23e-12` represent?
26. Can floats be used to store numbers with unlimited precision?
27. What are the differences between integers and floats?
28. How do you convert an integer to a float?
29. How do you convert a float to an integer?
30. What is the result obtained when you convert 1.99 to an integer?
31. What are the data types of the results of the division operators `/` and `//`?
32. What kind of data does the Boolean data type represent?
33. Which types of Python operators return booleans as a result?
34. What happens if you try to use a boolean in arithmetic operation?
35. How can any value in Python be covered to a boolean?
36. What are truthy and falsy values?
37. What are the values in Python that evaluate to False?
38. Give some examples of values that evaluate to True.
39. What kind of data does the None data type represent?
40. What is the purpose of None?
41. What kind of data does the String data type represent?
42. What are the different ways of creating strings in Python?
43. What is the difference between strings creating using single quotes, i.e. `'` and `'` vs. those created using double quotes, i.e. `"` and `"`?
44. How do you create multi-line strings in Python?
45. What is the newline character, `\n`?
46. What are escaped characters? How are they useful?
47. How do you check the length of a string?
48. How do you convert a string into a list of characters?
49. How do you access a specific character from a string?
50. How do you access a range of characters from a string?
51. How do you check if a specific character occurs in a string?
52. How do you check if a smaller string occurs within a bigger string?
53. How do you join two or more strings?

54. What are "methods" in Python? How are they different from functions?
55. What do the `.lower`, `.upper` and `.capitalize` methods on strings do?
56. How do you replace a specific part of a string with something else?
57. How do you split the string "Sun,Mon,Tue,Wed,Thu,Fri,Sat" into a list of days?
58. How do you remove whitespace from the beginning and end of a string?
59. What is the string `.format` method used for? Can you give an example?
60. What are the benefits of using the `.format` method instead of string concatenation?
61. How do you convert a value of another type to a string?
62. How do you check if two strings have the same value?
63. Where can you find the list of all the methods supported by strings?
64. What is a list in Python?
65. How do you create a list?
66. Can a Python list contain values of different data types?
67. Can a list contain another list as an element within it?
68. Can you create a list without any values?
69. How do you check the length of a list in Python?
70. How do you retrieve a value from a list?
71. What is the smallest and largest index you can use to access elements from a list containing five elements?
72. What happens if you try to access an index equal to or larger than the size of a list?
73. What happens if you try to access a negative index within a list?
74. How do you access a range of elements from a list?
75. How many elements does the list returned by the expression `a_list[2:5]` contain?
76. What do the ranges `a_list[:2]` and `a_list[2:]` represent?
77. How do you change the item stored at a specific index within a list?
78. How do you insert a new item at the beginning, middle, or end of a list?
79. How do you remove an item from al list?
80. How do you remove the item at a given index from a list?
81. How do you check if a list contains a value?
82. How do you combine two or most lists to create a larger list?
83. How do you create a copy of a list?
84. Does the expression `a_new_list = a_list` create a copy of the list `a_list`?
85. Where can you find the list of all the methods supported by lists?
86. What is a Tuple in Python?
87. How is a tuple different from a list?
88. Can you add or remove elements in a tuple?
89. How do you create a tuple with just one element?
90. How do you convert a tuple to a list and vice versa?
91. What are the `count` and `index` method of a Tuple used for?
92. What is a dictionary in Python?
93. How do you create a dictionary?
94. What are keys and values?
95. How do you access the value associated with a specific key in a dictionary?
96. What happens if you try to access the value for a key that doesn't exist in a dictionary?
97. What is the `.get` method of a dictionary used for?
98. How do you change the value associated with a key in a dictionary?
99. How do you add or remove a key-value pair in a dictionary?
100. How do you access the keys, values, and key-value pairs within a dictionary?