Python Programming for Data Science

Chapter 1: Python Basics

Chapter Outline

Chapter Learning Objectives

- Create, describe and differentiate standard Python datatypes such as `int`, `float`, `string`, `list`, `dict`, `tuple`, etc.
- Perform arithmetic operations like `+`, `-`, `*`, `**` on numeric values.
- Perform basic string operations like `.lower()`, `.split()` to manipulate strings.
- Compute boolean values using comparison operators operations (`==`, `!=`, `>`, etc.) and boolean operators (`and`, `or`, `not`).
- Assign, index, slice and subset values to and from tuples, lists, strings and dictionaries.
- Write a conditional statement with `if`, `elif` and `else`.
- Identify code blocks by levels of indentation.
- Explain the difference between mutable objects like a `list` and immutable objects like a `tuple`.

1. Introduction

The material presented on this website assumes no prior knowledge of Python. Experience with programming concepts or another programming language will help, but is not required to understand the material.

The website comprises the following:

1. **Chapters**: these contain the core content. Read through these at your leisure.
2. **Practice Exercises**: there are optional practice exercise sets to complement each chapter (solutions included). Try your hand at these for extra practice and to help solidify concepts in the **Chapters**.

2. Basic Python Data Types

A **value** is a piece of data that a computer program works with such as a number or text. There are different **types** of values: `42` is an integer and `"Hello!"` is a string. A **variable** is a name that refers to a value. In mathematics and statistics, we usually use variable names like x and y. In Python, we can use any word as a variable name as long as it starts with a letter or an underscore. However, it should not be a reserved word in Python such as `for`, `while`, `class`, `lambda`, etc. as these words encode special functionality in Python that we don't want to overwrite!

It can be helpful to think of a variable as a box that holds some information (a single number, a vector, a string, etc). We use the **assignment operator** `=` to assign a value to a variable.

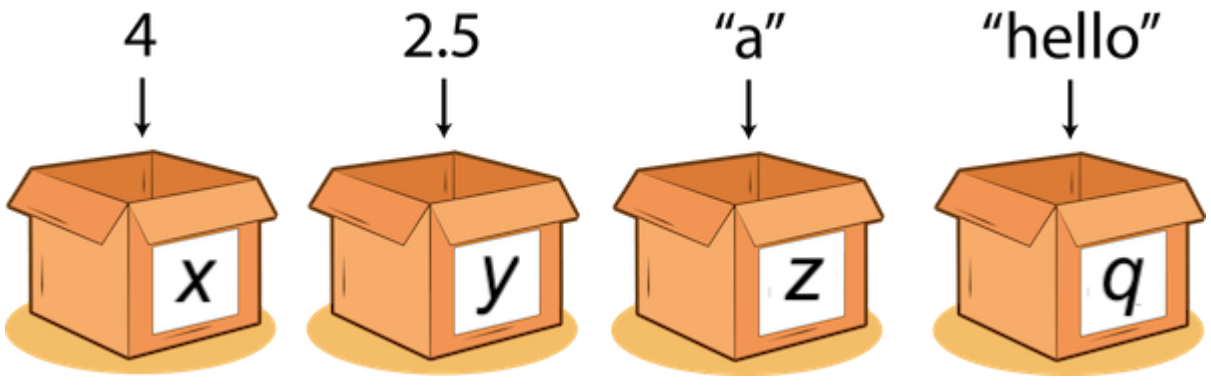

Image modified from: medium.com

Tip

See the Python 3 documentation for a summary of the standard built-in Python datatypes.

Common built-in Python data types

English name	Type name	Type Category	Description	Example
integer	`int`	Numeric Type	positive/negative whole numbers	`42`
floating point number	`float`	Numeric Type	real number in decimal form	`3.14159`
boolean	`bool`	Boolean Values	true or false	`True`
string	`str`	Sequence Type	text	`"I Can Has Cheezburger?"`
list	`list`	Sequence Type	a collection of objects - mutable & ordered	`['Ali', 'Xinyi', 'Miriam']`
tuple	`tuple`	Sequence Type	a collection of objects - immutable & ordered	`('Thursday', 6, 9, 2018)`
dictionary	`dict`	Mapping Type	mapping of key-value pairs	`{'name':'DSCI', 'code':511, 'credits':2}`
none	`NoneType`	Null Object	represents no value	`None`

Numeric data types

There are three distinct numeric types: `integers`, `floating point numbers`, and `complex numbers` (not covered here). We can determine the type of an object in Python using `type()`. We can print the value of the object using `print()`.

```
x = 42
```

```
type(x)
```

```
int
```

```
print(x)
```

```
42
```

In Jupyter/IPython (an interactive version of Python), the last line of a cell will automatically be printed to screen so we don't actually need to explicitly call `print()`.

```
x  # Anything after the pound/hash symbol is a comment and will not be run
```

```
42
```

```
pi = 3.14159
pi
```

```
3.14159
```

```
type(pi)
```

```
float
```

Arithmetic Operators

Below is a table of the syntax for common arithmetic operations in Python:

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation
//	integer division / floor division
%	modulo

Let's have a go at applying these operators to numeric types and observe the results.

```
1 + 2 + 3 + 4 + 5  # add
```

```
15
```

```
2 * 3.14159  # multiply
```

```
6.28318
```

```
2 ** 10  # exponent
```

```
1024
```

Division may produce a different `dtype` than expected, it will change `int` to `float`.

```
int_2 = 2
type(int_2)
```

```
int
```

```
int_2 / int_2  # divison
```

```
1.0
```

```
type(int_2 / int_2)
```

```
float
```

But the syntax `//` allows us to do “integer division” (aka “floor division”) and retain the `int` data type, it always rounds down.

```
101 / 2
```

```
50.5
```

```
101 // 2  # "floor division" - always rounds down
```

```
50
```

We refer to this as “integer division” or “floor division” because it’s like calling `int` on the result of a division, which rounds down to the nearest integer, or “floors” the result.

```
int(101 / 2)
```

```
50
```

The `%` “modulo” operator gives us the remainder after division.

```
100 % 2  # "100 mod 2", or the remainder when 100 is divided by 2
```

```
0
```

```
101 % 2  # "101 mod 2", or the remainder when 101 is divided by 2
```

```
1
```

```
100.5 % 2
```

```
0.5
```

None

`NoneType` is its own type in Python. It only has one possible value, `None` - it represents an object with no value. We’ll see it again in a later chapter.

```
x = None
```

```
print(x)
```

```
None
```

```
type(x)
```

```
NoneType
```

Strings

Text is stored as a data type called a `string`. We can think of a string as a sequence of characters.

> **Tip**
>
> Actually they are a sequence of Unicode code points. Here's a great blog post on Unicode if you're interested.

We write strings as characters enclosed with either:

- single quotes, e.g., `'Hello'`
- double quotes, e.g., `"Goodbye"`

There's no difference between the two methods, but there are cases where having both is useful (more on that below)! We also have triple double quotes, which are typically used for function documentation (more on that in a later chapter), e.g., `"""This function adds two numbers"""`.

```
my_name = "Tomas Beuzen"
```

```
my_name
```

```
'Tomas Beuzen'
```

```
type(my_name)
```

```
str
```

```
course = 'DSCI 511'
```

```
course
```

```
'DSCI 511'
```

```
type(course)
```

```
str
```

If the string contains a quotation or apostrophe, we can use a combination of single and double quotes to define the string.

```
sentence = "It's a rainy day."
```

```
sentence
```

```
"It's a rainy day."
```

```
type(sentence)
```

```
str
```

```
quote = 'Donald Knuth: "Premature optimization is the root of all
evil."'
```

```
quote
```

```
'Donald Knuth: "Premature optimization is the root of all evil."'
```

Boolean

The Boolean (`bool`) type has two values: `True` and `False`.

```
the_truth = True
```

```
the_truth
```

```
True
```

```
type(the_truth)
```

```
bool
```

```
lies = False
```

```
lies
```

```
False
```

```
type(lies)
```

```
bool
```

Comparison Operators

We can compare objects using comparison operators, and we'll get back a Boolean result:

Operator	Description
`x == y`	is `x` equal to `y`?
`x != y`	is `x` not equal to `y`?
`x > y`	is `x` greater than `y`?
`x >= y`	is `x` greater than or equal to `y`?
`x < y`	is `x` less than `y`?
`x <= y`	is `x` less than or equal to `y`?
`x is y`	is `x` the same object as `y`?

```
2 < 3
```

```
True
```

```
"Deep learning" == "Solve all the world's problems"
```

```
False
```

```
2 != "2"
```

```
True
```

```
2 is 2
```

```
True
```

```
2 == 2.0
```

```
True
```

Boolean Operators

We also have so-called “boolean operators” which also evaluates to either `True` or `False`:

Operator	Description
`x and y`	are `x` and `y` both True?
`x or y`	is at least one of `x` and `y` True?
`not x`	is `x` False?

```
True and True
```

```
True
```

```
True and False
```

```
False
```

```
True or False
```

```
True
```

```
False or False
```

```
False
```

```
("Python 2" != "Python 3") and (2 <= 3)
```

```
True
```

```
True
```

```
True
```

```
not True
```

```
False
```

```
not not True
```

```
True
```

> **Note**
>
> Python also has bitwise operators like `&` and `|`. Bitwise operators literally compare the bits of two integers. That's beyond the scope of this course but I've included a code snippet below to show you them in action.

```
print(f"Bit representation of the number 5: {5:0b}")
print(f"Bit representation of the number 4: {4:0b}")
print(f"                                    ↓↓↓")
print(f"                                    {5 & 4:0b}")
print(f"                                     ↓ ")
print(f"                                     {5 & 4}")
```

```
Bit representation of the number 5: 101
Bit representation of the number 4: 100
                                    ↓↓↓
                                    100
                                     ↓
                                     4
```

Casting

Sometimes we need to explicitly **cast** a value from one type to another. We can do this using functions like `str()`, `int()`, and `float()`. Python tries to do the conversion, or throws an error if it can't.

```
x = 5.0
type(x)
```

```
float
```

```
x = int(5.0)
x
```

```
5
```

```
type(x)
```

```
int
```

```
x = str(5.0)
x
```

```
'5.0'
```

```
type(x)
```

```
str
```

```
str(5.0) == 5.0
```

```
False
```

```
int(5.3)
```

```
5
```

```
float("hello")
```

```
---------------------------------------------------------------------
------
ValueError                                Traceback (most recent call
last)
<ipython-input-60-7124e8e12e61> in <module>
----> 1 float("hello")

ValueError: could not convert string to float: 'hello'
```

3. Lists and Tuples

Lists and tuples allow us to store multiple things ("elements") in a single object. The elements are *ordered* (we'll explore what that means a little later). We'll start with lists. Lists are defined with square brackets `[]`.

```
my_list = [1, 2, "THREE", 4, 0.5]
```

```
my_list
```

```
[1, 2, 'THREE', 4, 0.5]
```

```
type(my_list)
```

```
list
```

Lists can hold any datatype - even other lists!

```
another_list = [1, "two", [3, 4, "five"], True, None, {"key": "value"}]
another_list
```

```
[1, 'two', [3, 4, 'five'], True, None, {'key': 'value'}]
```

You can get the length of the list with the function `len()`:

```
len(my_list)
```

```
5
```

Tuples look similar to lists but have a key difference (they are immutable - but more on that a bit later). They are defined with parentheses `()`.

```
today = (1, 2, "THREE", 4, 0.5)
```

```
today
```

```
(1, 2, 'THREE', 4, 0.5)
```

```
type(today)
```

```
tuple
```

```
len(today)
```

```
5
```

Indexing and Slicing Sequences

We can access values inside a list, tuple, or string using square bracket syntax. Python uses *zero-based indexing*, which means the first element of the list is in position 0, not position 1.

```
my_list
```

```
[1, 2, 'THREE', 4, 0.5]
```

```
my_list[0]
```

```
1
```

```
my_list[2]
```

```
'THREE'
```

```
len(my_list)
```

```
5
```

```
my_list[5]
```

```
---------------------------------------------------------------------
------
IndexError                                Traceback (most recent call
last)
<ipython-input-74-075ca585e721> in <module>
----> 1 my_list[5]

IndexError: list index out of range
```

We can use negative indices to count backwards from the end of the list.

```
my_list
```

```
[1, 2, 'THREE', 4, 0.5]
```

```
my_list[-1]
```

```
0.5
```

```
my_list[-2]
```

```
4
```

We can use the colon `:` to access a sub-sequence. This is called "slicing".

```
my_list[1:3]
```

```
[2, 'THREE']
```

Note from the above that the start of the slice is inclusive and the end is exclusive. So `my_list[1:3]` fetches elements 1 and 2, but not 3.

Strings behave the same as lists and tuples when it comes to indexing and slicing. Remember, we think of them as a *sequence* of characters.

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
```

```
alphabet[0]
```

```
'a'
```

```
alphabet[-1]
```

```
'z'
```

```
alphabet[-3]
```

```
'x'
```

```
alphabet[:5]
```

```
'abcde'
```

```
alphabet[12:20]
```

```
'mnopqrst'
```

List Methods

A list is an object and it has methods for interacting with its data. A method is like a function, it performs some operation with the data, but a method differs to a function in that it is defined on the object itself and accessed using a period `.`. For example, `my_list.append(item)` appends an item to the end of the list called `my_list`. You can see the documentation for more list methods.

```
primes = [2, 3, 5, 7, 11]
primes
```

```
[2, 3, 5, 7, 11]
```

```
len(primes)
```

```
5
```

```
primes.append(13)
```

```
primes
```

```
[2, 3, 5, 7, 11, 13]
```

Sets

Another built-in Python data type is the `set`, which stores an *un-ordered* list of *unique* items. Being unordered, sets do not record element position or order of insertion and so do not support indexing.

```
s = {2, 3, 5, 11}
s
```

```
{2, 3, 5, 11}
```

```
{1, 2, 3} == {3, 2, 1}
```

```
True
```

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

```
False
```

```
s.add(2)  # does nothing
s
```

```
{2, 3, 5, 11}
```

```
s[0]
```

```
---------------------------------------------------------------------
------
TypeError                                 Traceback (most recent call
last)
<ipython-input-93-c9c96910e542> in <module>
----> 1 s[0]

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

Above: throws an error because elements are not ordered and can't be indexing.

Mutable vs. Immutable Types

Strings and tuples are immutable types which means they can't be modified. Lists are mutable and we can assign new values for its various entries. This is the main difference between lists and tuples.

```
names_list = ["Indiana", "Fang", "Linsey"]
names_list
```

```
['Indiana', 'Fang', 'Linsey']
```

```
names_list[0] = "Cool guy"
names_list
```

```
['Cool guy', 'Fang', 'Linsey']
```

```
names_tuple = ("Indiana", "Fang", "Linsey")
names_tuple
```

```
('Indiana', 'Fang', 'Linsey')
```

```
names_tuple[0] = "Not cool guy"
```

```
---------------------------------------------------------------------
------
TypeError                                 Traceback (most recent call
last)
<ipython-input-97-bd6a1b77b220> in <module>
----> 1 names_tuple[0] = "Not cool guy"

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

Same goes for strings. Once defined we cannot modifiy the characters of the string.

```
my_name = "Tom"
```

```
my_name[-1] = "q"
```

```
---------------------------------------------------------------------
------
TypeError                                 Traceback (most recent call
last)
<ipython-input-99-9bfcf81dbcf0> in <module>
----> 1 my_name[-1] = "q"

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

```
x = ([1, 2, 3], 5)
```

```
x[1] = 7
```

```
---------------------------------------------------------------------
------
TypeError                                 Traceback (most recent call
last)
<ipython-input-101-415ce6bd0126> in <module>
----> 1 x[1] = 7

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

```
x
```

```
([1, 2, 3], 5)
```

```
x[0][1] = 4
```

```
x
```

```
([1, 4, 3], 5)
```

4. String Methods

There are various useful string methods in Python.

```
all_caps = "HOW ARE YOU TODAY?"
all_caps
```

```
'HOW ARE YOU TODAY?'
```

```
new_str = all_caps.lower()
new_str
```

```
'how are you today?'
```

Note that the method lower doesn't change the original string but rather returns a new one.

```
all_caps
```

```
'HOW ARE YOU TODAY?'
```

There are *many* string methods. Check out the documentation.

```
all_caps.split()
```

```
['HOW', 'ARE', 'YOU', 'TODAY?']
```

```
all_caps.count("O")
```

```
3
```

One can explicitly cast a string to a list:

```
caps_list = list(all_caps)
caps_list
```

```
['H',
 'O',
 'W',
 ' ',
 'A',
 'R',
 'E',
 ' ',
 'Y',
 'O',
 'U',
 ' ',
 'T',
 'O',
 'D',
 'A',
 'Y',
 '?']
```

```
"".join(caps_list)
```

```
'HOW ARE YOU TODAY?'
```

```
"-".join(caps_list)
```

```
'H-O-W- -A-R-E- -Y-O-U- -T-O-D-A-Y-?'
```

We can also chain multiple methods together (more on this when we get to NumPy and Pandas in later chapters):

```
"".join(caps_list).lower().split(" ")
```

```
['how', 'are', 'you', 'today?']
```

String formatting

Python has ways of creating strings by “filling in the blanks” and formatting them nicely. This is helpful for when you want to print statements that include variables or statements. There are a few ways of doing this but I use and recommend f-strings which were introduced in Python 3.6. All you need to do is put the letter “f” out the front of your string and then you can include variables with curly-bracket notation `{}`.

```
name = "Newborn Baby"
age = 4 / 12
day = 10
month = 6
year = 2020
template_new = f"Hello, my name is {name}. I am {age:.2f} years old. I
was born {day}/{month:02}/{year}."
template_new
```

```
'Hello, my name is Newborn Baby. I am 0.33 years old. I was born
10/06/2020.'
```

Note

Notes require **no** arguments, In the code above, the notation after the colon in my curly braces is for formatting. For example, `:.2f` means, print this variable with 2 decimal places. See format code options here.

5. Dictionaries

A dictionary is a mapping between key-values pairs and is defined with curly-brackets:

```
house = {
    "bedrooms": 3,
    "bathrooms": 2,
    "city": "Vancouver",
    "price": 2499999,
    "date_sold": (1, 3, 2015),
}

condo = {
    "bedrooms": 2,
    "bathrooms": 1,
    "city": "Burnaby",
    "price": 699999,
    "date_sold": (27, 8, 2011),
}
```

We can access a specific field of a dictionary with square brackets:

```
house["price"]
```

```
2499999
```

```
condo["city"]
```

```
'Burnaby'
```

We can also edit dictionaries (they are mutable):

```
condo["price"] = 5  # price already in the dict
condo
```

```
{'bedrooms': 2,
 'bathrooms': 1,
 'city': 'Burnaby',
 'price': 5,
 'date_sold': (27, 8, 2011)}
```

```
condo["flooring"] = "wood"
```

```
condo
```

```
{'bedrooms': 2,
 'bathrooms': 1,
 'city': 'Burnaby',
 'price': 5,
 'date_sold': (27, 8, 2011),
 'flooring': 'wood'}
```

We can also delete fields entirely (though I rarely use this):

```
del condo["city"]
```

```
condo
```

```
{'bedrooms': 2,
 'bathrooms': 1,
 'price': 5,
 'date_sold': (27, 8, 2011),
 'flooring': 'wood'}
```

And we can easily add fields:

```
condo[5] = 443345
```

```
condo
```

```
{'bedrooms': 2,
 'bathrooms': 1,
 'price': 5,
 'date_sold': (27, 8, 2011),
 'flooring': 'wood',
 5: 443345}
```

Keys may be any immutable data type, even a `tuple`!

```
condo[(1, 2, 3)] = 777
condo
```

```
{'bedrooms': 2,
 'bathrooms': 1,
 'price': 5,
 'date_sold': (27, 8, 2011),
 'flooring': 'wood',
 5: 443345,
 (1, 2, 3): 777}
```

You'll get an error if you try to access a non-existent key:

```
condo["not-here"]
```

```
---------------------------------------------------------------------
------
KeyError                                  Traceback (most recent call
last)
<ipython-input-126-ab081f66baa5> in <module>
----> 1 condo["not-here"]

KeyError: 'not-here'
```

6. Empties

Sometimes you'll want to create empty objects that will be filled later on.

```
lst = list()  # empty list
lst
```

```
[]
```

```
lst = []  # empty list
lst
```

```
[]
```

There's no real difference between the two methods above, `[]` is apparently marginally faster...

```
tup = tuple()  # empty tuple
tup
```

```
()
```

```
tup = ()  # empty tuple
tup
```

```
()
```

```
dic = dict()  # empty dict
dic
```

```
{}
```

```
dic = {}  # empty dict
dic
```

```
{}
```

```
st = set()  # empty set
st
```

```
set()
```

7. Conditionals

Conditional statements allow us to write programs where only certain blocks of code are executed depending on the state of the program. Let's look at some examples and take note of the keywords, syntax and indentation.

```
name = "Tom"

if name.lower() == "tom":
    print("That's my name too!")
elif name.lower() == "santa":
    print("That's a funny name.")
else:
    print(f"Hello {name}! That's a cool name!")
print("Nice to meet you!")
```

```
That's my name too!
Nice to meet you!
```

The main points to notice:

- Use keywords `if`, `elif` and `else`
- The colon `:` ends each conditional expression
- Indentation (by 4 empty space) defines code blocks
- In an `if` statement, the first block whose conditional statement returns `True` is executed and the program exits the `if` block
- `if` statements don't necessarily need `elif` or `else`
- `elif` lets us check several conditions
- `else` lets us evaluate a default block if all other conditions are `False`
- the end of the entire `if` statement is where the indentation returns to the same level as the first `if` keyword

If statements can also be **nested** inside of one another:

```
name = "Super Tom"

if name.lower() == "tom":
    print("That's my name too!")
elif name.lower() == "santa":
    print("That's a funny name.")
else:
    print(f"Hello {name}! That's a cool name.")
    if name.lower().startswith("super"):
        print("Do you really have superpowers?")

print("Nice to meet you!")
```

```
Hello Super Tom! That's a cool name.
Do you really have superpowers?
Nice to meet you!
```

Inline if/else

We can write simple `if` statements "inline", i.e., in a single line, for simplicity.

```
words = ["the", "list", "of", "words"]

x = "long list" if len(words) > 10 else "short list"
x
```

```
'short list'
```

```
if len(words) > 10:
    x = "long list"
else:
    x = "short list"
```

```
x
```

```
'short list'
```

Truth Value Testing

Any object can be tested for "truth" in Python, for use in `if` and `while` (next chapter) statements.

- `True` values: all objects return `True` unless they are a `bool` object with value `False` or have `len()` == 0
- `False` values: `None`, `False`, `0`, empty sequences and collections: `''`, `()`, `[]`, `{}`, `set()`

Tip

Read more in the docs here.

```
x = 1

if x:
    print("I'm truthy!")
else:
    print("I'm falsey!")
```

```
I'm truthy!
```

```
x = False

if x:
    print("I'm truthy!")
else:
    print("I'm falsey!")
```

```
I'm falsey!
```

```
x = []

if x:
    print("I'm truthy!")
else:
    print("I'm falsey!")
```

```
I'm falsey!
```

Short-circuiting

Python supports a concept known as "short-circuting". This is the automatic stopping of the execution of boolean operation if the truth value of expression has already been determined.

```
fake_variable  # not defined
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-142-38b1451e4717> in <module>
----> 1 fake_variable  # not defined

NameError: name 'fake_variable' is not defined
```

```
True or fake_variable
```

```
True
```

```
True and fake_variable
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-144-a7196cc665d5> in <module>
----> 1 True and fake_variable

NameError: name 'fake_variable' is not defined
```

```
False and fake_variable
```

```
False
```

Expression	Result	Detail
A or B	If A is `True` then A else B	B only executed if A is `False`
A and B	If A is `False` then A else B	B only executed if A is `True`