Introduction to Programming in Python

Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Python Basics

Numbers

Math with integers works as you would expect:

In [ ]:
```
5 + 2
```

In [ ]:
```
2 - 4
```

In [ ]:
```
7 * (6 + 1) # brackets work as usual
```

In [ ]:
```
2 ** 3 # two to the third power
```

All in one cell:

In [ ]:

```
5 + 2
2 - 4
7 * (6 + 1)
2 ** 3
```

What happened? Jupyter will only print the result of the *last* expression in a cell. We can fix that by using the `print` function:

In [ ]:

```
print(5 + 2)
print(2 - 4)
print(7 * 7)
print(2 ** 3)
```

What about division?

In [ ]:

```
2 / 3
```

This works too, but it returns a different kind of number: a floating point number or `float`. This is true even when the division could in principle be done exactly:

In [ ]:

```
6 / 2
```

To a human, 3.0 (a `float`) and 3 (an integer or `int`) represent the same number, they are represented differently in memory; we say that these two objects have a different **type**. We can find the type of an object like this:

In [ ]:

```
type(3)
```

In [ ]:

```
type(3.0)
```

Math with floats can be a bit tricky, because they are represented with finite precision, which means that not all numbers are representable:

In [ ]:
```
1 - 0.9
```

Variables

A variable is a named memory location. It is assigned using "`=`"

In [ ]:

```
a = 2
b = 4
c = a + b
print(c)
```

Easy enough. Can you guess what the following does?

In [ ]:

```
a = 2
a = a + 1
print(a)
```

The code below is equivalent:

In [ ]:

```
a = 2
a += 1  # shorthand for a = a + 1
print(a)
```

Variable names can be made up from letters, numbers, and the underscore. They may not start with a number. Python is case-sensitive: `A` is not the same as `a`.

Assignment versus equality

We just saw that variables are assigned using `=`.

In [ ]:

```
a = 3
print(a)
```

What if we want to compare if two numbers are equal? First attempt:

In [ ]:

```
# uncomment the next line and run the cell
# 3 = 3
```

This obviously didn't work. The correct way is to use `==`:

In [ ]:
```
3 == 3
```

In [ ]:
```
1 == 2
```

The returned object is of type `bool` (a "Boolean"):

In [ ]:
```
type(True)
```

A `bool` can take one of two values: `True` or `False`.

They are returned by *relational operators*: `<`, `<=`, `>`, `>=`, `==` (equality), `!=` (inequality), and can be combined using the *logical operators* `and`, `or`, and `not`.

In [ ]:
```
1 <= 2 < 4
```

In [ ]:
```
1 < 2 and 2 < 1
```

In [ ]:
```
not(1 < 2)
```

Strings

Strings hold text. They are constructed using either single or double quotes:

In [ ]:
```
s1 = "Python"
s2 = ' is easy.'
s3 = s1 + s2  # Concatenation
```

In [ ]:
```
type(s3)
```

This doesn't work:

In [ ]:
```
a = 3 # an int
b = "4" # a string
# uncomment and run:
# a  + b
```

We have to convert the string first:

In [ ]:
```
a + int(b)
```

We can also convert the other way:

In [ ]:
```
a = 3
b = str(a)
```

In [ ]:
```
type(b)
```

This is useful for printing:

In [ ]:
```
height = 1.89
print("I am " + str(height) + "m tall.")
```

One way to obtain a string is to ask the user for input:

In [ ]:

```
mystr = input("What's your name? ")
print(mystr)
```

Exercise

Write some code in the cell below that asks the user for their age, and then prints the age in dog years (i.e., divided by 7).

Example input:

```
What's your age?
```

If the user enters `28`, then this should result in the following output:

```
Your age in dog years is 4.0.
```

Note that `input` always returns a string, so you have to convert it to `int` (or `float`) to do math with it.

In [ ]:

```

```

Sequence Types: Containers with Integer Indexing

Strings

We have already encountered `string`s: they hold text, and are constructed using single or double quotes:

In [ ]:

```
s1 = "Python"; s2 = ' is easy.'; s3 = s1 + s2  # Concatenation
print(s3)
```

The `len` function returns the length of a string:

In [ ]:

```
len(s3)
```

We can select characters from a string by *indexing* into it:

In [ ]:
```
print(s3)
```

In [ ]:
```
s3[0]  # Note zero-based indexing
```

In [ ]:
```
s3[-1]  # Negative indexes count from the right:
```

We can also pick out several elements ("*slicing*"). This works for all *sequence types* (lists, NumPy arrays, ...).

In [ ]:
```
print(s3)
```

In [ ]:
```
s3[0:2] # Elements 0 and 1; left endpoint is included, right endpoint excluded.
```

In [ ]:
```
s3[0:6:2] # start:stop:step
```

In [ ]:
```
s3[::-1] # start and stop can be ommitted; default to 0 and len(str)
```

Exercise

Use slicing to extract the substring `sign` from the string `s` below.

In [ ]:

```
s = 'Why did I sign up for this?'
```

Methods for strings

Apart from functions (like `len` and `print`), Python also has *methods*. They work almost like functions, but they are called by appending the name of the method to the name of an object:

In [ ]:

```
s1 = "Simon"
```

In [ ]:

```
s1.upper()
```

Objects of different types support different operations (methods). Here is a list for strings:

In [ ]:
```
print(', '.join(filter(lambda m: callable(getattr(s1, m)) and not m.startswith("_"), dir(s1))))
```

The key in the command above is `dir(s1)` (try it). The rest is just for pretty printing.

Another way to find out which methods are supported by an object of a given type is to use *autocompletion*:

In [ ]:
```
# uncomment the next line
# s1. # try hitting the tab key after the dot
```

You can find out what a method does by using the `help` facility:

In [ ]:
```
help(s1.upper)
```

Exercise

The following methods for strings are needed in the homeworks. Try to find out what they do, using a mix of `help` and try and error: `lower`, `upper`, `capitalize`, `startswith`, `endswith`, `index`, and `find`.

In [ ]:

```

```

Lists

Lists are indexable collections of arbitrary (though usually homogeneous) things:

In [ ]:
```
list1 = [1, 2., 'hi']; print(list1)
```

As for strings, the function `len` returns the length of a list (or any other sequence):

In [ ]:
```
len(list1)
```

Like strings, they support indexing, but unlike strings, they are *mutable*, i.e., elements of the list can be changed

In [ ]:
```
list1 = [1, 2., 'hi']
list1[2] = 42
print(list1)
```

The `sum`, `min`, and `max` functions respectively compute the sum, minumum, and maximum of a list, provided this is meaningful considering the elements of the list:

In [ ]:
```
list1 = [1, 2., 42]
print(sum(list1))
print(min(list1))
print(max(list1))
```

Like strings, lists support a number of methods:

In [ ]:

```
print(', '.join(filter(lambda m: callable(getattr(list1, m)) and not m.startswith("_"), dir(list1
```

Exercise

The `append`, `insert`, `index`, `remove`, `reverse`, and `count` methods are needed for the homeworks. Find out what they do.

In [ ]:

```
```

Tuples

- A `tuple` is similar to a list, but *immutable*. It is created with round brackets:

In [ ]:
```
(1, 2., 'hi')
```

Exercise

1. Create a list containing the names "Simon", "Carl", and "Lucy" as strings, and store it in a variable.
2. Change the second element of the list to "Karl".
3. Repeat, but now using a tuple instead of a list. Why does this fail?

```
In [ ]:
```

Other built-in datatypes

- Other built-in datatypes include `set`s (unordered collections) and `dict`s (collections of key-value pairs). More on these later.

Homework

Beginner exercises 1-4, 6-7, 9-10, 18 from https://holypython.com/