Lesson 1

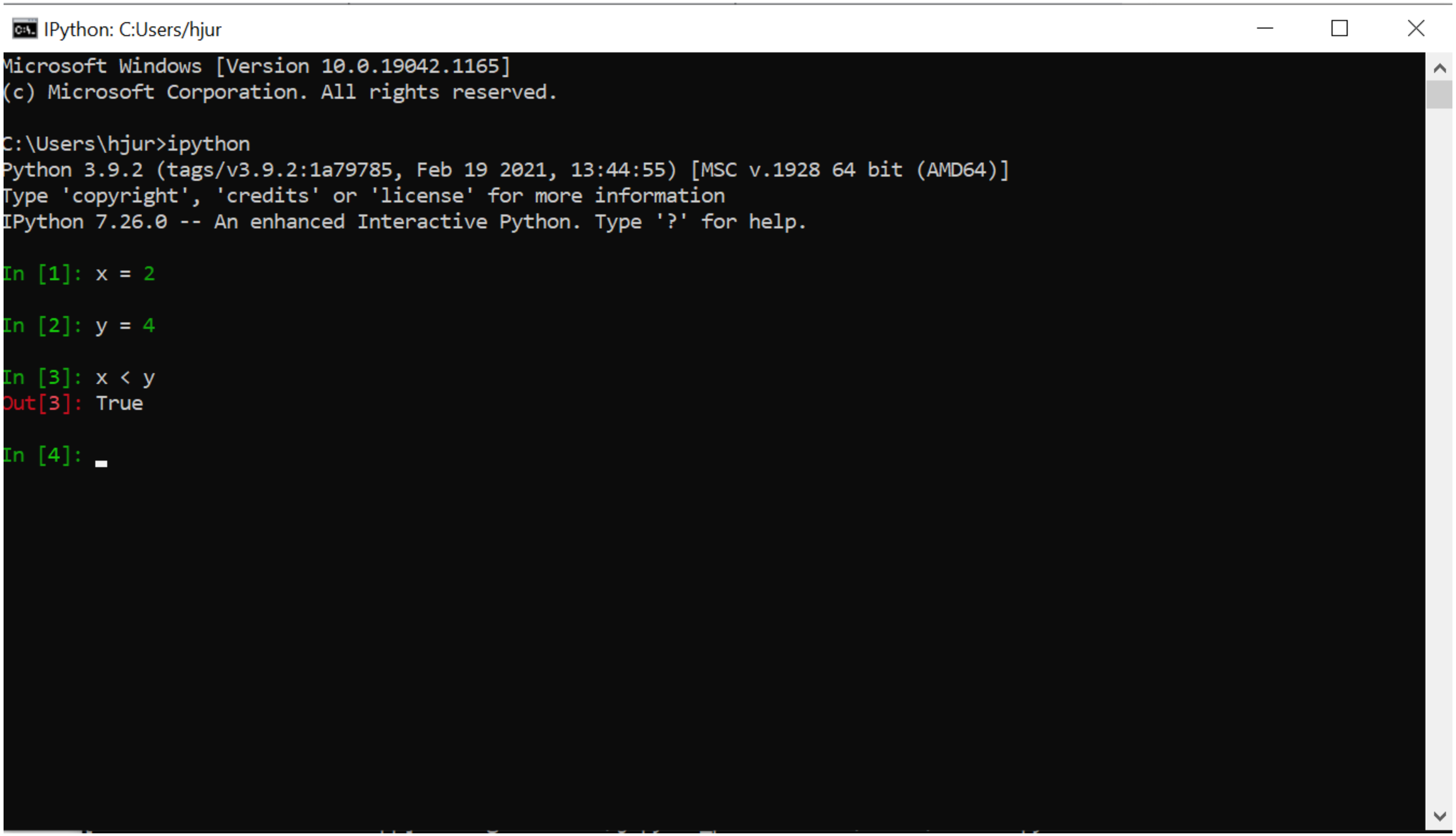
Anaconda, IPython, Jupyter notebooks, Python code, variables, integers, floats, calculations

Anaconda

Anaconda is a distribution of the Python programming languages for scientific computing that aims to simplify package management and deployment. The distribution includes data-science packages suitable for Windows, Linux, and macOS. We will use it a lot and we recommend you to download it. Anaconda comes with a number of applications that we will use such as Jupiter Notebook and the IPython console. You can download Anaconda at: https://www.anaconda.com/

IPython

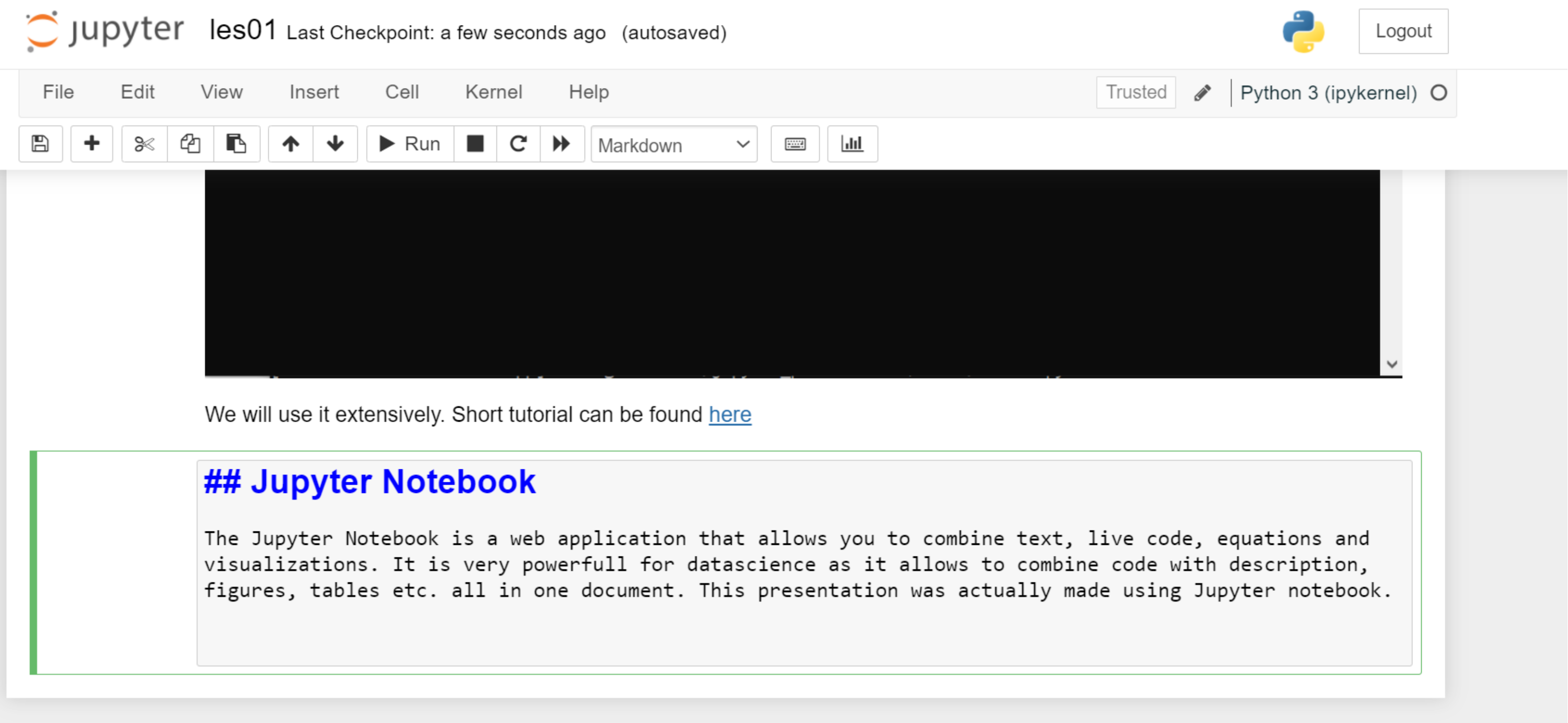
We will use IPython for interactive execution of Python code. It is much more versatile compared to Python default interactive envirenment and we highly recommend you to use it.



We will use it extensively. Short tutorial can be found here

Jupyter Notebook

The Jupyter Notebook is a web application that allows you to combine text, live code, equations and visualizations. It is very powerfull for datascience as it allows to combine code with description, figures, tables etc. all in one document. This presentation was actually made using Jupyter notebook:



Variables

- All programs need data to work on.
- To store data, you need a placeholder for it. These are called variables .

Here are a few examples:

Variables

- All programs need data to work on.
- To store data, you need a placeholder for it. These are called variables .

Here are a few examples:

In [1]:
```
amino_acid = "alanine"
number_of_atoms = 13
mw = 89.09
```

Here you can find a movie that explains variables.

Information (data) comes in many forms, such as numbers, characters, words, pictures, sound. In programming languages like python information is stored in variables. You can think of a box that has a label on it and stuff is stored into that. You can find or use the information later by searching for the box with the label you stored the information in.

What happens when I type:

What happens when I type:

In [2]:
```
amino_acid = "alanine"
```

What happens when I type:

```python
In [2]: amino_acid = "alanine"
```

- amino_acid is the name of the variable you have just created
- The assignment (=) operator was used to assign the string alanine to the variable amino_acid
- 'alanine' is a literal ,the value to be assigned to the variable amino_acid

Python has several data types. We will cover several types in this lesson:

- Integers
- Floats

Integers

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

Integers

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

In [3]:

```
x = 3
y = -3

print(type(x))
print(type(y))
```

```
<class 'int'>
<class 'int'>
```

- *The function* `type` *shows the datatype: int or integer in this case.*
- *The function* `print` *is used to print data to the screen.*

Floats

Floats represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10. Some examples:

Floats

Floats represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10. Some examples:

In [4]:

```
x = 2.5
y = 12E64

print(type(x))
print(type(y))
```

```
<class 'float'>
<class 'float'>
```

Basic calculations with Python

Of course, Python (being a programming language) supports calculations.
Here are some examples:

Basic calculations with Python

Of course, Python (being a programming language) supports calculations.
Here are some examples:

In [5]:
```
x = 6
y = 3
print(x + y)
print(x - y)
print(x * y)
print(x / y)
```

```
9
3
18
2.0
```

Basic calculations with Python

Of course, Python (being a programming language) supports calculations.
Here are some examples:

In [5]:
```
x = 6
y = 3
print(x + y)
print(x - y)
print(x * y)
print(x / y)
```

```
9
3
18
2.0
```

Note that the devision changes the datatype from an integer to a float!

Other calculations

Some other (though still basic) calculations might be a bit less obvious. For example exponentiation:

Other calculations

Some other (though still basic) calculations might be a bit less obvious. For example exponentiation:

In [6]:

```
x = 3
print(x**3)
```

```
27
```

> *You might be tempted to use ^ (just as Excel) but this will NOT give the expected result. ^ is not covered here.*

Floor devision is another frequently used operator. Floor division is a normal division operation but it returns the largest possible integer (chopping the decimal part). It is (for example) convenient to use to calculate how many hours there are in 134 minutes:

Floor devision is another frequently used operator. Floor division is a normal division operation but it returns the largest possible integer (chopping the decimal part). It is (for example) convenient to use to calculate how many hours there are in 134 minutes:

In [7]:

```
x = 134
print(x // 60)
```

```
2
```

The modulo operator (%) is another frequently used operator. It calculates the left over after a devision. For example 17 % 5 = 2. It is (for example) convenient to switch 134 minutes to hours:minutes notation:

The modulo operator (%) is another frequently used operator. It calculates the left over after a devision. For example 17 % 5 = 2. It is (for example) convenient to switch 134 minutes to hours:minutes notation:

In [8]:

```
x = 134
print( x % 60)
```

```
14
```

Or in total: convert 134 minutes to hour:minutes notation:

Or in total: convert 134 minutes to hour:minutes notation:

In [9]:

```python
x = 134
print(x // 60)
print(x % 60)

# bit of formatting you will learn later:

print(x // 60, ":", x % 60, sep="")
```

```
2
14
2:14
```

Getting help

Python comes with great help built-in. You can use the `help` funtion to get help on funtions, data objects etc. Here an example for help on the print function:

In [10]:

```python
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Here we only add the name of the function (print) as an argument in the help function. We don't execute it. So this will not work to get help on the print function:

In [11]:

```
help(print()) # not the same as the previous example. Will be explained at a future lesson.
```

```
Help on NoneType object:

class NoneType(object)
 |  Methods defined here:
 |
 |  __bool__(self, /)
 |      self != 0
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
```

But you can also get help on a self declared variable:

In [12]:

```python
x = 3
help(x)
```

```
Help on int object:

class int(object)
 |  int([x]) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Built-in subclasses:
 |      bool
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
```

```
 |  __and__(self, value, /)
 |      Return self&value.
 |
 |  __bool__(self, /)
 |      self != 0
 |
 |  __ceil__(...)
 |      Ceiling of an Integral returns itself.
 |
 |  __divmod__(self, value, /)
 |      Return divmod(self, value).
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __float__(self, /)
 |      float(self)
 |
 |  __floor__(...)
 |      Flooring an Integral returns itself.
 |
 |  __floordiv__(self, value, /)
 |      Return self//value.
 |
 |  __format__(self, format_spec, /)
 |      Default object formatter.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getnewargs__(self, /)
```

```
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __index__(self, /)
 |      Return self converted to an integer, if self is suitable for use as an in
dex into a list.
 |
 |  __int__(self, /)
 |      int(self)
 |
 |  __invert__(self, /)
 |      ~self
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __lshift__(self, value, /)
 |      Return self<<value.
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mod__(self, value, /)
 |      Return self%value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
```

```
 |
 |  __neg__(self, /)
 |      -self
 |
 |  __or__(self, value, /)
 |      Return self|value.
 |
 |  __pos__(self, /)
 |      +self
 |
 |  __pow__(self, value, mod=None, /)
 |      Return pow(self, value, mod).
 |
 |  __radd__(self, value, /)
 |      Return value+self.
 |
 |  __rand__(self, value, /)
 |      Return value&self.
 |
 |  __rdivmod__(self, value, /)
 |      Return divmod(value, self).
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __rfloordiv__(self, value, /)
 |      Return value//self.
 |
 |  __rlshift__(self, value, /)
 |      Return value<<self.
 |
 |  __rmod__(self, value, /)
 |      Return value%self.
 |
```

```
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  __ror__(self, value, /)
 |      Return value|self.
 |
 |  __round__(...)
 |      Rounding an Integral returns itself.
 |      Rounding with an ndigits argument also returns an integer.
 |
 |  __rpow__(self, value, mod=None, /)
 |      Return pow(value, self, mod).
 |
 |  __rrshift__(self, value, /)
 |      Return value>>self.
 |
 |  __rshift__(self, value, /)
 |      Return self>>value.
 |
 |  __rsub__(self, value, /)
 |      Return value-self.
 |
 |  __rtruediv__(self, value, /)
 |      Return value/self.
 |
 |  __rxor__(self, value, /)
 |      Return value^self.
 |
 |  __sizeof__(self, /)
 |      Returns size in memory, in bytes.
 |
 |  __sub__(self, value, /)
 |      Return self-value.
 |
```

```
 |  __truediv__(self, value, /)
 |      Return self/value.
 |
 |  __trunc__(...)
 |      Truncating an Integral returns itself.
 |
 |  __xor__(self, value, /)
 |      Return self^value.
 |
 |  as_integer_ratio(self, /)
 |      Return integer ratio.
 |
 |      Return a pair of integers, whose ratio is exactly equal to the original i
nt
 |      and with a positive denominator.
 |
 |      >>> (10).as_integer_ratio()
 |      (10, 1)
 |      >>> (-10).as_integer_ratio()
 |      (-10, 1)
 |      >>> (0).as_integer_ratio()
 |      (0, 1)
 |
 |  bit_length(self, /)
 |      Number of bits necessary to represent self in binary.
 |
 |      >>> bin(37)
 |      '0b100101'
 |      >>> (37).bit_length()
 |      6
 |
 |  conjugate(...)
 |      Returns self, the complex conjugate of any int.
 |
```

```
 |  to_bytes(self, /, length, byteorder, *, signed=False)
 |      Return an array of bytes representing an integer.
 |
 |      length
 |        Length of bytes object to use.  An OverflowError is raised if the
 |        integer is not representable with the given number of bytes.
 |      byteorder
 |        The byte order used to represent the integer.  If byteorder is 'big',
 |        the most significant byte is at the beginning of the byte array.  If
 |        byteorder is 'little', the most significant byte is at the end of the
 |        byte array.  To request the native byte order of the host system, use
 |        `sys.byteorder' as the byte order value.
 |      signed
 |        Determines whether two's complement is used to represent the integer.
 |        If signed is False and a negative integer is given, an OverflowError
 |        is raised.
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  from_bytes(bytes, byteorder, *, signed=False) from builtins.type
 |      Return the integer represented by the given array of bytes.
 |
 |      bytes
 |        Holds the array of bytes to convert.  The argument must either
 |        support the buffer protocol or be an iterable object producing bytes.
 |        Bytes and bytearray are examples of built-in objects that support the
 |        buffer protocol.
 |      byteorder
 |        The byte order used to represent the integer.  If byteorder is 'big',
 |        the most significant byte is at the beginning of the byte array.  If
 |        byteorder is 'little', the most significant byte is at the end of the
 |        byte array.  To request the native byte order of the host system, use
 |        `sys.byteorder' as the byte order value.
```

```
 |      signed
 |        Indicates whether two's complement is used to represent the integer.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  denominator
 |      the denominator of a rational number in lowest terms
 |
 |  imag
 |      the imaginary part of a complex number
 |
 |  numerator
 |      the numerator of a rational number in lowest terms
 |
 |  real
 |      the real part of a complex number
```

The `dir` function is very convenient to explore what `methods` and `properties` are associated with any Python object. This will be explained more thoroughly at a later stage but let's just look at an example:

In [13]:

```python
x = 10
dir(x)
```

Out[13]:

```
['__abs__',
 '__add__',
 '__and__',
 '__bool__',
 '__ceil__',
 '__class__',
 '__delattr__',
 '__dir__',
 '__divmod__',
 '__doc__',
 '__eq__',
 '__float__',
 '__floor__',
 '__floordiv__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__index__',
 '__init__',
 '__init_subclass__',
 '__int__',
 '__invert__',
 '__le__',
 '__lshift__',
 '__lt__',
 '__mod__',
```

```
'__mul__',
'__ne__',
'__neg__',
'__new__',
'__or__',
'__pos__',
'__pow__',
'__radd__',
'__rand__',
'__rdivmod__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rfloordiv__',
'__rlshift__',
'__rmod__',
'__rmul__',
'__ror__',
'__round__',
'__rpow__',
'__rrshift__',
'__rshift__',
'__rsub__',
'__rtruediv__',
'__rxor__',
'__setattr__',
'__sizeof__',
'__str__',
'__sub__',
'__subclasshook__',
'__truediv__',
'__trunc__',
'__xor__',
'as_integer_ratio',
```

```
'bit_length',
'conjugate',
'denominator',
'from_bytes',
'imag',
'numerator',
'real',
'to_bytes']
```

This all might seem a bit overwelming but with a little practice it will soon be clear. Just remember that you can get help using the `help` function and that you can "inspect" any Python object using the `dir` function.

The end...