Programming with Python

May 4, 2017

Python and IPython

Python is an *interpreter*. It translates *source code* into instructions that the processor can understand.
Different ways of running python:

python

```
$ python
Python 2.7.3 (default, Mar 13 2014, 11:03:55)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license"
for more information.
>>>

>>> print("Hello␣Python␣World!")
Hello Python World!
>>>
```

Python reads your input, evaluates the command and prints a result.
Just like the shell it is a `Read-Eval-Print Loop`, short *REPL*.

Python and IPython

Interactive Python is another REPL, which is very similar to *MATLAB* or *Mathematica*.

ipython

```
$ ipython
Python 2.7.3 (default, Mar 13 2014, 11:03:55)
Type "copyright", "credits" or "license" for more information.

IPython 3.0.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for
             extra details.

In [1]: print("Hello␣Python␣World!")
Hello Python World!

In [2]:
```

Python and IPython

Python commands can be put in a text file and executed like a shell script.

greetings.py

```
print("Hello Python World!")
print("This works as a script as well!")
```

```
$ python greetings.py
Hello Python World!
This works as a script as well!
```

Python language basics

Now we know how to run python programs. Let's learn the basics of Python!

Comments

It is good practice to put comments in your code. Document what a certain piece of code does. Python uses the # character to denote comments. Lines starting with # are ignored by the interpreter.

```
# this line is a comment
print("Hello comments")      # This will print "Hello comments"
```

Python language basics

Variables

Variables have a name and a value. To assign a value to a name the equals sign (=) is used.
Variable names can contain upper and lower case letters, numbers and underscores (_).
CAUTION: variable name cannot start with a digit.

```
h_bar = 1.05457e-34
pi = 3.1415926
```

Once variables have a value assigned to them, we can use them in calculations.

```
h = 2 * pi * h_bar
print(h)
```

Python language basics

All variables in Python have a type, that is dynamically assigned to the variable when it is created.

Basic data types

```
dims = 3                        # int, only digits
ndim = 3.0                      # float, because of the '.'
h_bar = 1.05457e-34             # float because of the ',' or 'e'
label = "Energy (in MeV)" # str, text surrounded by quotes
```

Integers and strings are *precise* types.
Float are *imprecise*, because of how floating point numbers are represented in the computer. Usually, they are 64-Bit *approximations* to real numbers. Beware of rounding issues.
You can ask Python for the type of a variable:

```
In [1]: type(h_bar)
Out[1]: float

In [2]: type(42)
Out[2]: int
```

Python language basics

You can use the type names to convert between types.

Converting types

```
In [1]: float(42)
Out[1]: 42.0

In [2]: int("28")
Out[2]: 28
```

The second one, of course, only works if the string contains digits only. It would produce an error otherwise.

Python language basics

Variables in Python are dynamically typed.

- types are set to value of the variable, not the name
- variable types do not need to be known before variable is used
- variable names can change type, when their values change

Dynamic typing

```
x = 3                    # x is now an int
x = 3.1415926            # x is now a float
x = "Energy (in MeV)" # x is now a string
```

Other languages such as C and FORTRAN are *statically* typed. Type of variable cannot change and needs to be known before using the variable.

Python language basics

Special Variables

Python has some special built-in variables. These unique variables are known as *singletons*.

Boolean Values

```
True
False
```

These can be used for logical expressions. Other data types can be converted to booleans. Basically, if the variable is zero or a container is empty it converts to `False`. If the value is non-zero on non-empty it converts to `True`.

Python language basics

Special Variables

None

`None` is not Zero! `None` is used to denote that no value was assigned. Which is different from 0.

NotImplemented

`NotImplemented` differs from `None` it that way, that it denotes an action that is impossible to executed, not only non-existent.

Python language basics

Arithmetic Operators

Let's assume we have two variables `a = 10` and `b = 20`:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -10
*	Multiplication - Multiplies values on either side of the operator	a * b will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 0
**	Exponent - Performs exponential (power) calculation on operators	a**b will give 10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 is equal to 4 and 9.0//2.0 is equal to 4.0

Python language basics

Comparison Operators

Let's assume we have two variables `a = 10` and `b = 20`:

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(a == b) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.

Python language basics

Assignment Operators

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	c = a + b will assigne value of a + b into c
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**=	Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//=	Floor Dividion and assigns a value, Performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Python language basics

Bitwise Operators

Let's assume `a = 60` and `b = 13` which is `a = 0011 1100` and `b = 0000 1101` in binary:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(a & b) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in eather operand.	(a | b) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(a ^ b) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the efect of 'flipping' bits.	(~a) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	a << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 will give 15 which is 0000 1111

Python language basics

Logical Operators

Let's assume we have two variables `a = 10` and `b = 20`:

Operator	Description	Example
and	Called Logical AND operator. If both the operands are true then then condition becomes true.	(a and b) is true.
or	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(a or b) is true.
not	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	not(a and b) is false.

Python language basics

Strings

Strings are a fundamental data type in Python. `str` is the type name, which can be used to convert other types into strings. For example `str(42)` will return `"42"`. Defining *string literals* can be done using matching single (') or double (") quotes.

```
line1 = "Nothing happens until something moves"
line2 = '- Albert Einstein'
```

Python strings are in general encoded in unicode UTF-8. That means you can use things like umlauts and Asian characters.

Python language basics

String indexing

Indexing means retrieving data from a part of the the string. Can be applied to all *sequences* in Python, in particular arrays of numbers, later.
In Python we use the square braces (`[]`) to operate on such a variable.

```
In [1]: p = "proton"

In [2]: p[1]
Out[2]: 'r'
```

Python uses *zero-indexing*, that means element count starts with 0. So the second element of the string `p` can be accessed using the index `1`.

Python language basics

String indexing

Indexing also works using *negative indices*. Element `-1` is the last element, `-2` is the second to last, and so on.

```
In [1]: p = "proton"

In [2]: p[-1]
Out[2]: 'n'
```

This is actually a shortcut for `p[len(p)-1]`, where `len(p)` is a function that returns the lengths of a given string. But you do not need to write this explicitly all the time. One of the many nice things in Python!

Python language basics

String slicing

But what if we want to access more than one element at a time? You can extract substrings using a *slice*. A slice is a sequence-independent way to define a range of indices. The simplest form are two integers separated by a colon: `s[start:stop]`

```
In [3]: p[2:5]
Out[3]: 'oto'
```

Note that the `n` which actually is `p[5]` is not included. Slices are defined to include the lower bound, but exclude the upper bound. Or to express it mathematically, a slice is a half-open interval `[start, stop)`.

Python language basics

More slicing

For slicing the zero-indexing plays an important role. The difference between the start and stop values will always be the length of the slice. To express this in code, the following expression is always true for any given sequence `s`:

```
(start - stop) == len(s[start:stop])
```

You can also freely mix positive and negative indices when slicing.

```
In [4]: p[1:-1]
Out[4]: 'roto'

In [5]: p[-1:2]
Out[5]: ''
```

But a slice cannot wrap around the edges of a sequence, this would return an empty string.

Python language basics

More slicing

Another great Python feature is that the start and stop values are optional. If you omit one or both the default values will be used. Namely, start becomes zero and stop becomes the length of the list. The colon still has to be present though.

```
s[:2]    # the first two elements
s[-5:]   # the last five elements
s[:]     # the whole string, equivalent to s
```

Python language basics

More slicing

Slicing has another parameter: the *step* otherwise also known as *stride*. The step is a number, that represents how many elements to go in the sequence before picking up the next element. So for example you only want to pick every second or third element of a list.

Thus the full notation for slicing is given by:

`s[start:stop:step]`

```
In [1]: q = "AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz"
In [2]: q[2:-2:2]
Out[2]: 'BCDEFGHIJKLMNOPQRSTUVWXY'

In [3]: q[1::2]
Out[3]: 'abcdefghijklmnopqrstuvwxyz'

In [4]: q[::-3]
Out[4]: 'zYwVtSqPnMkJhGeDbA'
```

Python language basics

String concatenation

Strings can be manipulated using operators. The easiest one we want to consider is the (+) operator. It is used to glue strings together, which is known as *concatenation*.

```
In [1]: "kilo" + "meter"
Out[1]: 'kilometer'
```

Other data types can be converted to string and concatenated with another string:

```
In [1]: "x^" + str(2)
Out[1]: 'x^2'
```

Give that addition on strings works and multiplication (*) is many additions, multiplying strings works as expected:

```
In [1]: "Hello" * 8
Out[1]: 'HelloHelloHelloHelloHelloHelloHelloHello'
```

Only addition and multiplication work on strings. Strings cannot be subtracted, divided or exponentiated.

Python language basics

String literals

Two string literals which are next to each other will be stuck together automatically.

```
In [1]: "H + H"         " -> H2"
Out[1]: 'H + H -> H2'
```

Newlines between parentheses are ignored. So long strings can span multiple lines.

```
quote = ("Science is what we understand well enough to explain
  to a computer."
  "Art is everything else we do."
  " - Donald Knuth")
```

If you need a single or a double quote to be part of a string, use the other kind to define the string at the outermost level.

```
x = "It's easy as this!"
y = 'The computer said, "Permission denied"'
```

Python language basics

String literals

If you need both quote type inside your string you have to use *escape* characters.

```
"Bones said, \"He\'s dead, Jim.\""
```

Char-acter	Interpretation
\\	Backslash
\n	Newline - start a new line
\r	Carriage return - go to the start of *this* line.
\t	Tab
\'	Single quote
\"	Double quote

Special string escape characters.

Python language basics

Multiline strings

Lastly, Python supports multiline strings, that preserve newlines. These strings need to be surrounded with either triple single quotes (’’’) or triple double quotes (""").

```
"""THIS is thy hour O Soul, thy free flight into the wordless,
Away from books, away from art, the day erased, the lesson done,
Thee fully forth emerging, silent, gazing, pondering the themes
thou lovest best.
Night, sleep, death and the stars
"""
```

Python language basics

String methods

Variables in Python may have other variables that "live in" them. These are known as *attributes*. Attributes are accessed using the dot operator (`.`) . Some attributes are *functions* and are then called *methods*. This is due to the object-oriented nature of the Python language, which we will cover most likely later.

One particularly useful string method is `strip()`. It removes all leading and trailing whitespaces of a string, while preserving internal whitespaces. A whitespace it either a space, tab, newline, or other blank character.

```
In [1]: header = "    temperature  pressure\t value \n"
In [2]: header.strip()
Out[2]: 'temperature  pressure\t value'
```

Python language basics

String methods

The methods `upper()` and `lower()` will return a string with all upper case or lower case letters, respectively.

```
In [3]: header.upper()
Out[3]: header = "    TEMPERATURE  PRESSURE\t VALUE \n"
```

The method `swapcase()` will switch the existing case. The `isdigit()` method returns `True` or `False` if the string contains only integers or not.

```
In [4]: "10".isdigit()
Out[4]: True
In [4]: "10.10".isdigit()
Out[4]: False
```

Python language basics

String methods

Finally, the `format()` method. It creates new strings from templates with the template values filled in. The basic syntax for a template is an integer in curly braces ({}).

```
In [1]: "{0} gets into work and then
         his {1} begins.".format("Hilbert", "commute")
Out[1]: 'Hilbert gets into work and then his commute begins.'
```

You can use template to convert data types to strings on-the-fly.

```
In [1]: x = 42
In [2]: y = 65.0
In [3]: "x={0} y={1}".format(x, y)
Out[3]: 'x=42 y=65.0'

In [4]: "x=" + str(x) + " y=" str(y)
Out[4]: 'x=42 y=65.0'
```

Essential Containers

Python comes with a number of data containers built-in.
These are date types that are used to hold other variables. Each container has its own type and properties.
Here we will cover the following data types: `list`, `tuple`, `set` and `dict`.

Essential Containers

Mutability

A data type is considered *mutable* if its values are allowed to change – if its values have *state*. It is called *immutable* if its values are static and unchangeable once it is created. Immutable data can be used to created new variables based on existing values, but not change the original value.

The data types we covered so far `int`, `float`, `bool` and `str` are immutable. It does not make sense to change the value of `1`.

`1` is `1` and so integers are immutable.

Containers are partially defined by whether they are mutable or not.

Essential Containers

Lists

Lists in Python are one-dimensional, ordered containers. The elements of such a list can be any Python object. Lists are mutable and have methods to add and remove elements from themselves. The literal syntax for a list are comma separated values surrounded with square braces.

```
[6, 28]
[1e3, -2, "I am a list element"]
[[1.0, 0.0], [0.0, 1.0]]
```

The type of elements do not have to match unlike in other programming languages.

Essential Containers

Lists

List can be concatenated together with the addition operator (+) to make a longer list.

```
In [1]: [1, 1] + [2, 3, 5] + [8]
Out[1]: [1, 1, 2, 3, 5, 8]
```

You can add single elements to the end of a list using the `append()` method.

```
In [2]: fib = [1, 1, 2, 3, 5, 8]
In [3]: fib.append(13)
In [4]: fib
Out[4]: [1, 1, 2, 3, 5, 8, 13]
```

Essential Containers

Lists

Multiple elements can be added to the list using the `extend()` method or the (+=) operator.

```
In [5]: fib.extend([21, 34, 55])
In [6]: fib
Out[6]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
In [7]: fib += [89, 144]
In [8]: fib
Out[8]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

List indexing works exactly the same as string indexing.

```
In [9]: fib[::2]
Out[9]: [1, 2, 5, 13, 34, 89]
```

Essential Containers

Lists

Other neat list tricks:

```
In [10]: fib[3] = "whoops" # set fourth element to 'whoops'
In [11]: fib
Out[11]: [1, 1, 2, 'whoops', 5, 8, 13, 21, 34, 55, 89, 144]

In [12]: del fib[:5] # remove the first five elements of fib
In [13]: fib
Out[13]: [8, 13, 21, 34, 55, 89, 144]

In [14]: fib[1::2] = [-1, -1, -1] # Assign -1 to each odd element
In [15]: fib
Out[15]: [8, -1, 21, -1, 55, -1, 144]
```

Essential Containers

Lists

Some more list tricks:

The multiplication-by-an-integer trick also works for lists:

```
In [1]: [1, 2, 3] * 6
Out[1]: [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

A list of can be created from a string using the `list()` data type converter.

```
In [1]: list("F = dp/dt")
Out[1]: ['F', ' ', '=', ' ', 'd', 'p', '/', 'd', 't']

In [1]: x = [] # creates an empty list
```

Essential Containers

Tuples

Tuples are the immutable form of lists.
They behave the same way as lists do, but once created you cannot change their value anymore. There are no `append()` or `extend()` methods.
Syntax differs as well. Tuples are defined by commas `(,)`.
The parentheses are optional and only included for readability.

```
a = 1, 2, 5, 3    # tuple of length 4
b = (42,)         # tuple of length 1, defined by the comma
c = (42)          # not a tuple, but an int
d = ()            # length-0 tuple, no commas mean no elements
```

Essential Containers

Tuples

You can concatenate tuples together, just as lists or strings.
But this creates a new tuple!
The order of operations is important here, this is where the parentheses come in handy.

```
In [1]: (1, 2) + (3, 4)
Out[1]: (1, 2, 3, 4)

In [2]: 1, 2 + 3, 4
Out[2]: 1, 5, 4
```

You can use the type name `tuple()` to convert lists into tuples.

```
In [1]: tuple(["e", 2,718])
Out[1]: ('e', 2.718)
```

Essential Containers

Tuples

Even though tuples are immutable, they can have mutable elements.
So if you have a list as an element of a tuple, you can modify that list just as before. But you cannot remove the list completely from the tuple.

```
In [2]: x = 1.0, [2, 4], 16

In [3]: x[1].append(8)

In [4]: x
Out[4]: (1.0, [2, 4, 8], 16)
```

Essential Containers

Lists vs. Tuples

Other than mutability, what are the differences between lists and tuples?

In principle there are none. In practice they are used for different purposes. Even though there are no strict rules about this the common usage is as follows.
Lists are used for *homogeneous* data (all elements `int`, `str`, etc.).
Tuples on the other hand are used for *heterogeneous* data `('C14', 6, 14, 14.00324198843)`.

Tuples are integral to functions, as we will see later.
Tuples as well as lists can have duplicate elements.
The next data type we are looking at ensures unique elements!

Essential Containers

Sets

The Python type `set` is equivalent to mathematical sets.
Sets are defined by comma separated values between curly braces {}, just like their math counterparts. Sets are unordered containers of unique values. Duplicate entry will be ignored. Since they are unordered, sets cannot be indexed. Containment, such as 'is `x` in `y`' is in the focus and not how the elements are stored.

```
# a literal set with elements of different types
{1.0, 10, "one hundred", (1, 0, 0.0)}

# a literal set of special values
{True, False, None, "", 0.0, 0}

# conversion from a list to a set
set([2.0, 4, 'eight', (16,)])
```

Essential Containers

Sets

How does the set of a single string look like?

The set of a string is actually the set of its characters, because strings are sequences.

```
In [1]: set("Marie Curie")
Out[1]: {' ', 'C', 'M', 'a', 'e', 'i', 'r', 'u'}

In [2]: set(["Marie Curie"])
Out[2]: {'Marie Curie'}
```

Essential Containers

Sets

Set operators

Let's consider two sets `s = {1, 2, 3}` and `t = {3, 4}`.

Operation	Meaning	Example
`s | t`	Union	`{1, 2, 3, 4, 5}`
`s & t`	Intersection	
`s - t`	Difference - elements in `s` but not in `t`	`{1, 2}`
`s ^ t`	Symmetric difference - elements not in `s` or `t`	`{1, 2, 4, 5}`
`s < t`	Strict Subset	`False`
`s <= t`	Subset	`False`
`s > t`	Strict Superset	`False`
`s >= t`	Superset	`False`

Essential Containers

Sets

The uniqueness of the elements of a set is the key property. This defines a restriction of what can go in a set. Elements of a set must be *hashable*.

What is hashing?

Suppose there is a function that takes any value and maps it to an integer. If two variables map to the same integer they must have the same value. In Python such a function is simply called `hash()`. You can use it on any variable. If for some reason the function fails, that value cannot be placed in a set.

Essential Containers

Sets

What make a type hashable? Immutability!

Suppose you could hash a list. If you were to add an element to that list, its hash would change. You could now have two lists with the same elements in a set, which would brake uniqueness. Hence, lists are not allowed in sets. But tuples are, but only if all their elements are hashable.
Sets themselves are mutable. You can `add()` or `discard()` elements. If you need an immutable set you can convert it to a `frozenset()`.

Essential Containers

Dictionaries

Dictionaries are the *most important* data type is Python.
A dictionary - or `dict()` - is a mutable, unordered collection of unique key-value pairs.
Keys are associated with values. You can look up a value knowing only its keys. Keys must be unique. However, multiple keys can have the same value. Working with dictionaries is extremely fast and efficient.
Both keys and values are Python objects, with the restriction of hashability for the keys. They are defined by curly braces $\{\}$ surrounding key-value pairs separated by commas. Each key-value pair, is known as an *item* and they are separated by a colon.

Essential Containers

Dictionaries

Here are some examples:

```
# A dictionary on one line
al = {"first": "Albert", "last": "Einstein"}

# You can split up dicts on multiple lines
constants = {
  'pi': 3.1415926,
  'e': 2.718,
  'h': 6.62606957e-34
  True: 1.0,
}

# A dict can be created from a list of (key, value) tuples
axes = dict([(1, "x"), (2, "y"), (3, "z")])
```

Essential Containers

Dictionaries

To access values from a dictionary you 'index' it using its key.

```
In [1]: constants['e']
Out[1]: 2.718

In [2]: axes[3]
Out[2]: 'z'

In [3]: al['last']
Out[3]: 'Einstein'
```

Since dicts are unordered, slicing is impossible for them. However you can add or change values by indexing them.

```
constants[False] = 0.0 # adds a new constant
del axes[3]            # deletes third axes from dict
al['first'] = "Al"
```

Essential Containers

Dictionaries

Since dictionaries are mutable they are not hashable. You cannot use a dictionary as a key in another dictionary.

Because dictionaries were implemented before sets you created empty sets and dictionaries like this

```
{}     # empty dictionary
set() # empty set
```

Tests for containment with the `in` operator work on dictionary keys only.

```
In [3]: "N_A" in constants
Out[3]: False
```

More on dictionaries later . . .

Modules

Source code for Python is usually saved in files with the `.py` extension. When you load such a file into a Python interpreter it is called a *module*. A collection of modules in a directory is called a *package*. Python modules written in languages other than Python are possible and are then called *extension modules*. They are typically written in C or FORTRAN for speed!

Modules allow for grouping related code together and sharing code with other people.
Python has an extensive collection of very useful modules know as the *Python Standard Library*.

Modules

Importing Modules

To use modules in Python we need to use the *import* statement.
It has four different forms.
The first is just `import` followed by the module's name without the `.py` extension.

```
import <module>
```

Let' assume we have a file called `constants.py` with the following content:

```
pi = 3.1415926
h = 6.62606957e-64
```

In another file we can then import those constants and access them with the dot operator

```
import constants

two_pi = 2 * constants.pi
h_bar = constants.h / two_pi
```

Modules

Importing Variables from a Module

Writing `constants.var` all the time can be tedious if you have to use the same constant many times. This is where the `from-import` syntax steps in. You can import only a single or multiple variables from a module to use in your current file.

```
form <module> import <var>
from <module> import <var1>, <var2>, ...
```

This is a short form of the following

```
import <module>
<var> = <module>.<var>
del <module>
```

Modules

Importing Variables from a Module

Let' print out `constants.py` again:

```
pi = 3.1415926
h = 6.62606957e-64
```

For our example with the constants this changes to

```
from constants import pi, h

two_pi = 2 * pi
h_bar = h / two_pi
```

Modules

Aliasing Imports

The third way of importing allows to change the name of the imported module.
This can be useful if you already have a local variable of the same name that a module provides. This way uses the `as` keyword and has the following syntax: Let' print out `constants.py` again:

```
import <module> as <name>
```

which again is equivalent to the longer version

```
import <module>
<name> = <module>
del <module>
```

Modules

Aliasing Imports

Let's look at our example again.

```
# constants.py
pi = 3.1415926
h = 6.62606957e-64
```

Use constants with aliasing:

```
import constants as c

constants = 2.71828

two_pi = 2 * c.pi
h_bar = c.h / two_pi
```

`constants` in this example is a variable holding Euler's number, while the actual module in renamed to `c`.

Modules

Aliasing Variables on Import

The final form of importing combines from-importing and aliasing. You only import specific variables from a module renaming them in the process. Let' print out `constants.py` again:

```
from <module> import <var> as <name>
from <module> import <var1> as <name1>, <var2> as <name2>, ...
```

of course this again is equivalent to the longer version

```
from <module> import <var>
<name> = <var>
del <var>
```

Modules

Aliasing Variables on Import

With this out constants example looks like this:

```
# constants.py
pi = 3.1415926
h = 6.62606957e-64
```

And the other file:

```
from constants import pi as PI, h as H

two_pi = 2 * PI
h_bar = H / two_pi
```

Modules

Packages

Collections of modules in the same directory are called *packages*. To be visible to Python this directory must contain a special file name `__init__.py`. This is a signal to Python that this directory contains modules. The file can be empty, but if it contains code, this code will be executed before any modules are imported from that directory. A package can contain sub-packages (i.e. subdirectories). Each of them must contain their own `__init__.py`.

Modules

Packages

An example tree of a package with sub-packages.

```
compphys/
|-- __init__.py
|-- constants.py
|-- physics.py
|-- more/
|   |-- __init__.py
|   |-- morephysics.py
|   |-- evenmorephysics.py
|   |-- yetmorephysics.py
|-- raw/
|   |-- data.txt
|   |-- matrix.txt
|   |-- orphan.py
```

The package `compphys` has three modules (`__init__.py`, `constants.py` and `physics.py`) and one subpackage (`more`) with four modules. The `raw` directory is not a subpackage, because it lacks an __init__.py

Modules

Packages

Importing modules from packages uses the attribute access operator (`.`).
The syntax is the same as importing variables from a module.

```
import compphys.constants
import compphys.more.evenmorephysics

two_pi = 2 * compphys.constants.pi
```

Modules

Python Standard Library

Python comes with a huge library of tools, that make performing everyday task easy and pythonic. It includes platform-independent operating system tools, mathematical functions, compression algorithms, database access, and even a basic web server.

```
# small selection of Python standard modules
os       # operating system, file operations etc.
sys      # system specific functions
math     # functions and constants
re       # regular expressions
decimal  # arbitrary precision integers and floats
random   # pseudo-random number generators
csv      # tools for reading and writing comma separated values
```

On top of that come lots of third-party modules which support a rich and diverse environment for scientific computing and physics. (See `numpy`, `scipy`, and `sympy` later.)

Programming

Flow Control and Logic

Up to now, everything was very Python specific. The concepts we are about to learn in the next chapter are universal to all procedural programming languages although the syntax may differ.

Flow control is a high-level way of programming a computer to make decisions. Theses can be simple or complicated decisions that are executed once or multiple times. They determine the *execution pathway* for the program.

Python has *conditionals*, *exceptions*, and *loops*.

Flow Control and Logic

if-statements

Let's begin with conditionals. Conditionals are the simplest from of flow control. To express in English how conditionals work: "if x is true, then do something, otherwise do something else." The shortest conditional however is only an `if`-statement.

```
if <condition>:
    <if-block>
```

Here the keyword `if` is followed by an expression `condition` which itself is followed by a colon (`:`). When the boolean representation of the condition, `bool(condition)`, is `True`, then the code in the if block is executed, if it is `False` the code is skipped.

Flow Control and Logic

if-statements

For example, if we wanted to test if Planck's constant was equal to one and then change the value, we would write:

```
h_bar = 1.0
if h_bar == 1.0:
    print("This is not the right value for h_bar! Resetting...")
    h_bar = 1.05457173e-34
h = h_bar * 2 * 3.14159
```

Here `h_bar` would be reset to the real value, because we set it to `1.0` before. If it had been its physical value it would not have been reset. The crucial part of the `if`-statement in Python is that the `if`-block is indented by *four* whitespaces. Other languages use curly braces for example. Python uses indentation.
The last line is executed in any case of the expression above.

Flow Control and Logic

if-else-statements

Every `if`-statement may be follow by an optional `else`-statement. This is the keyword `else` followed by a colon at the same indentation level as the original `if`. The code block in the `else`-block is executed when the condition is `False`.

```
if <condition>:
    <if-block>
else:
    <else-block>
```

Flow Control and Logic

if-else-statements

For example let us consider the expression $\sin(1/x)$. This function is computable everywhere except for $x = 0$.
At this point, L'Hopital's rule shows the the result is also zero. We can use an `if-else`-statement to express this.

```
if x == 0:
    y = 0
else:
    y = sin(1/x)
```

We could also write

```
if x != 0:
    y = sin(1/x)
else:
    y = 0
```

However it is considered good practice to use positive conditionals (==) rather than negative ones (!=). Because humans tend to think about a condition to be true rather than it being false. This helps to eliminate logic bugs in your program.

Flow Control and Logic

if-elif-else-statements

Python also allows multiple optional `elif`-statements. The `elif` keyword is short for *else if*. Such statements come after the `if`-statement and before the final `else`-statement. The first conditional which evaluates to `True` determines which block is entered and executed.

```
if <condition0>:
    <if-block0>
elif <condition1>:
    <if-block1>
elif <condition2>:
    <if-block2>
...
else:
    <else-block>
```

Flow Control and Logic

if-elif-else-statements

Let us create a mid-band filter whose signal is 1 if the frequency is between 1 and 10Hz and zero otherwise.

```
if omega < 1.0:
    signal = 0.0
elif omega > 10.0:
    signal = 0.0
else:
    signal = 1.0
```

This could be extended to include ramping on either side of the band.

```
if omega < 0.9:
    signal = 0.0
elif omega > 0.9 and omega < 1.0:
    signal = (omega - 0.9) / 0.1
elif omega > 10.0 and omega < 10.1:
    signal = (10.1 - omega) / 0.1
elif omega > 10.1:
    signal = 0.0
else:
    signal = 1.0
```

Flow Control and Logic

if-else-expression

The final syntax covered here is the ternary conditional operator. It allows simple `if-else`-conditionals to be evaluated in a single expression.
This has the following syntax:

```
x if <condition> else y
```

If the condition is `True` `x` is returned. Otherwise `y` is returned.
Using this for our `h_bar` example from before:

```
h_bar = 1.05457173e-34 if h_bar == 1.0 else h_bar
```

This is similar to the `condition?x:y` operator from languages such as C, but way more readable.

Flow Control and Logic

Exceptions

Python, like most modern languages, has a mechanism for *exception handling*. This feature allows the programmer to deal with unexpected or catastrophic behavior of a program. The syntax for handling exceptions is known as a `try-except-block`. Both `try` and `except` are Python keywords. Try-excepts look very similar to if-else blocks without the condition,

```
try:
    <try-block>
except:
    <except-block>
```

The try-block will attempt to execute its code. If there are no errors the except-block is skipped and the program proceeds normally. If any error at all happens, the except block is entered immediately, no matter how far into the try block the program went. So keep try block as short as possible.

Flow Control and Logic

Exceptions

For example we can test for division by zero, which normally would crash a program.

```
try:
    inv = 1.0 / val
except:
    print("Bad value {0}, try again".format(val))
```

If `val` would be zero the program would crash on the second line without the try-except-block. The block allows us to catch the exception and handle it gracefully.

Flow Control and Logic

Exceptions

We can even specify the error we are anticipating to be caught. This allows for more specific behavior then a generic catch-all-exceptions.

```
try:
    inv = 1.0 / val
except ZeroDivisonError:
    print("A zero value was submitted, please try again!")
```

Multiple exceptions can be chained together, much like a elif-statement. The first exception that is called triggers its block.

```
try:
    inv = 1.0 / val
except ZeroDivisonError:
    print("A zero value was submitted, please try again!")
except:
    print("Bad value {0}, try again".format(val))
```

Flow Control and Logic

Raising Exceptions

You can also raise exceptions manually.
The `raise` keyword will *throw* and exception which may then be caught by a try-except-block elsewhere. Raise statements may appear anywhere, but is it common to put the inside of conditionals, so they are not executed unless really necessary.
Continuing with the division by zero example:

```
if val == 0.0:
    raise ZeroDivisionError
inv = 1.0 / val
```

If `val` happens to be zero, the last line would never be executed, because the exception would be raised before that.
You can specify a custom error message when raising exceptions, because sometimes a message "an error occurred" just isn't good enough.

```
if val == 0.0:
    raise ZeroDivisionError("inverse of zero not defined!")
inv = 1.0 / val
```

Flow Control and Logic

Exceptions

Python comes with 150+ error and exception types predefined.
Here is a small list of selected ones:

```
AssertionError     # used when assert operator returns False
AttributeError     # Python cannot find a variable that lives in an
IOError            # cannot read or write to a file
ImportError        # package or module cannot be found
KeyboardInterrupt # raised when user kills the program with Ctrl+C
MemoryError        # computer runs out of RAM
RuntimeError       # generic exception
SyntaxError        # raised when Python syntax is wrong
ZeroDivisionError # see above
```

Flow Control and Logic

Loops

So far we discussed single executions of indented code blocks. We use *loops* to execute the same block multiple times.
Python has a few loop formats that are essential to every Python programmer: `while`-loops, `for`-loops, and `comprehensions`.

Flow Control and Logic

while-loops

While loops are related to if-statements because the continue to execute 'while a condition is true'. The syntax is nearly identical, exchanging only the `if` and `while` keywords.

```
while <condition>:
    <while-block>
```

A simple countdown example:

```
t = 3
while t > 0:
    print("t-minus " + str(t))
    t = t - 1
print("blastoff!")
```

Flow Control and Logic

while-loops

If the condition is `False` the while-block will never be entered.

```
while False:
    print("I am sorry Dave.")
print("I can't print that for you")
```

If on the other hand the condition is always `True` the while-block will continue to execute forever.
This is known as an *infinite* or *non-terminating* loop.

DEMO

Flow Control and Logic

Breaking Loops

The `break`-statement is Python's way of leaving a loop early. The keyword simply appears on its own line and the loop is exited immediately.

For example this loop calculates the Fibonacci series and exits when it finds an entry that is divisible by 12.

```
fib = [1, 1]
while True:
    x = fib[-2] + fib[-1]
    if x % 12 == 0:
        break
    fib.append(x)
```

Flow Control and Logic

for-loops

Though while loops are helpful for repeating statements, it is sometimes more useful to iterate over a container or other iterable. This means taking a single element from a container, work through the block and continue doing so, until no more elements are in that container.
In Python this is realized with the keywords `for` and `in` with the following syntax:

```
for <loop-var> in <iterable>:
    <for-block>
```

The `loop-variable` is a variable name that is assigned to a new element of the iterable each time we pass through the loop.

Flow Control and Logic

for-loops

All containers (lists, tuples, sets, dicts) and strings are *iterable*.

Let us look at the countdown timer again. But this time with a `for`-loop.

```
for t in [3, 2, 1]:
    print("t-minus " + str(t))
print("blastoff!")
```

Flow Control and Logic

for-loops

In addition to the `break`-statement that exits a while-loop early, there is the `continue`-statement. It can be used in while and for-loops. This exits out of the current iteration and continues on to the next element. It does not exit the whole loop.
Let's consider another countdown where we want to skip even numbers.

```
for t in [7, 6, 5, 4, 3, 2, 1]:
    if t % 2 == 0:
        continue
    print("t-minus " + str(t))
print("blastoff!")
```

Flow Control and Logic

for-loops

Iterating over ordered and unordered data structures.

It is guaranteed that all elements are iterated over, but for unordered list the outcome is not predictable.

```
# works as expected, prints every letter
for letter in "Python":
    print(letter)

# iterating over a set might produce this output
# 0, True, String
for x in {"String", 0, True}:
    print(x)
```

Flow Control and Logic

for-loops

Dictionaries are even more complicated. The loop variable could be the keys, the values or even both (the items). By default Python chooses to return the keys, hence it is common to use `key` or `k` as loop variable.

```
d = {"first": "Albert",
     "last": "Einstein",
     "birthday": [1879, 3, 14]}

for key in d:
    print(key)
    print(d[key])
    print("====")
```

Flow Control and Logic

for-loops

But you can also explicitly loop over keys, values and items using the `keys()`, `values()` and `items()` methods.

```
d = {"first": "Albert",
     "last": "Einstein",
     "birthday": [1879, 3, 14]}

for key in d.keys():
    print(key)

for value in d.values():
    print(value)

for item in d.items():
    print(item)

# unpacking of items
for key, value in d.items():
    print(key, value)
```

Flow Control and Logic

for-loops

It is good programming practice and a strong Python idiom that the loop variable is a singular noun and the iterable is the corresponding plural noun. This makes loops easy to read and understand.

```
for single in plural:
    . . .
```

For example, looping through quark names in the Pythonic way:

```
quarks = {'up', 'down', 'top', 'bottom', 'charm', 'strange'}
for quark in quarks:
    print(quark)
```

Flow Control and Logic

Comprehensions

For and while loops are great, but they always take up at least to lines. One for the loop and at least one for the block. Often times we want to iterate over a container, perform some type of calculation and write the result back into a new container, which requires a third line.

```
quarks = {'up', 'down', 'top', 'bottom', 'charm', 'strange'}

upper_quarks = []
for quark in quarks:
    upper_quarks.append(quark.upper())
```

This can be done in one line, since there is only one meaningful operation in the loop, namely `upper_quarks.append(quark.upper())`.
Comprehensions to the rescue!

Flow Control and Logic

Comprehensions

Comprehensions are a syntax for simple for-loops in a single expression.
They work on lists, sets, and dicts.
The only limitation is, that the for-block is a single expression.

```
# List Comprehension
[<expr> for <loop-var> in <iterable>]

# Set Comprehension
{<expr> for <loop-var> in <iterable>}

# Dictionary Comprehension
{<key-expr>: <value-expr> for <loop-var> in <iterable>}
```

Flow Control and Logic

Comprehensions

Using comprehensions we can drastically shorten our quarks example, while still maintaining good readability.

```
quarks = {'up', 'down', 'top', 'bottom', 'charm', 'strange'}

upper_quarks = [quark.upper() for quark in quarks]
```

Flow Control and Logic

Comprehensions

Sometimes you might need a set comprehension instead of a list, for example to guarantee uniqueness of the entries. Let us consider quark name entries from a user in a list and put them in a set.

```
entries = ['top', 'CHARm', 'Top', 'STrange', 'stRAnGE', 'top']
quarks = {quark.lower() for quark in entries}

# result: quarks = {'top', 'charm', 'strange'}
```

Flow Control and Logic

Comprehensions

You can also write dictionary comprehensions. This can be useful, if you want to preform and expression over some data but retain a mapping from the input to the result. Suppose we want to create a dictionary that maps numbers from an `entries` list to the results of `x**2 + 42`. This can be done using:

```
entries = [1, 10, 12.5, 65, 88]
results = {x: x**2 + 42 for x in entries}
```

Flow Control and Logic

Comprehensions

Another powerful feature of comprehensions is the ability to put in a *filter*. This is a conditional entered after the iterable. If the condition evaluates to `True` the the loop expression is evaluated and added to the list, set or dictionary just as before. If the condition is `False` the iteration is skipped.
The syntax is as follows:

```
# List Comprehension with Filter
[<expr> for <loop-var> in <iterable> if <condition>]

# Set Comprehension with Filter
{<expr> for <loop-var> in <iterable> if <condition>}

# Dictionary Comprehension with Filter
{<key>: <value> for <loop-var> in <iterable> if <condition>}
```

Flow Control and Logic

Comprehensions

Let us compare the the list comprehension with filter to the long and explicit version.

```
# long version
new_list = []
for <loop-var> in <iterable>:
    if <condition>:
        new_list.append(<expr>)

# short version
new_list = [<expr> for <loop-var> in <iterable> if <condition>]
```

Flow Control and Logic

Comprehensions

Suppose you had a list of words `pm` that contains the entire text of Newton's *Principia Mathematica* and you want to find all words that start with the letter `t`. This can be done in one line using a comprehension with a filter:

```
t_words = [word for word in pm if word.startswith('t')]
```

Or you want to compute the set of squares of Fibonacci numbers `fib` if the number is divisible by five.

```
{x**2 for x in fib if x%5 == 0}
```

Flow Control and Logic

Comprehensions

Lastly, dictionary comprehensions with filter are most often used to retain or remove item from other dictionaries. Suppose we have a dictionary that maps coordinate axes to indices. We want to get only polar coordinates.

```
coords = {'x': 1, 'y': 2, 'z': 3, 'r': 4, 'theta': 5, 'phi': 6}
polar_keys = {'r', 'theta', 'phi'}
polar = {key: value for key, value in coords.items()
         if key in polar_keys}
```

Flow Control and Logic

Comprehensions

Comprehensions are incredibly powerful. But keep them short.
If an operation cannot be fit into a comprehension then it should be split up into a normal for-loop anyway. In principle comprehensions can be nested, but this can become pretty convoluted very quickly.
Again, Python is all about readability.
So use comprehensions where possible and use loops where necessary.

Functions

Software development is all about code reuse. (*Don't repeat yourself* principle).

Code reuse is great not only because you have to type less, but it also reduces the errors in your code. When performing the same sequence of operations over and over again they should be put into a *function*.
A function can be **called** as many times as desired. Calling a function executes all code inside that function. Sometimes an argument is **passed** into a function. Functions may or may not **return** values as their last operation.

Functions

Every programming language has its own way to define functions. In Python the first line of a function always starts with the `def`-keyword followed by the function name, the argument list in parentheses and a colon. The following lines are then indented like before and form the *body* of the function.

```
def <name>:
    <body>
```

The empty parentheses indicate that no parameters need to be passed to that function. Every function must have a *body*, thus the simplest function that does absolutely nothing is:

```
def nothing():
    pass
```

This is accomplished using the special `pass` statement.

Functions

Return values

Functions can return values using the `return` keyword. A function can have more that one return statement, but the function is exited after the first return statement is called. Let us consider a simple function that returns the number 42.

```
# define the function
def forty_two():
    return 42

# call the function
forty_two()

# call function and print result
print(forty_two())

# call function, assign result to x, print x
x = forty_two()
print(x)
```

Functions

Arguments

Like their math counterparts functions in Python may take arguments. These are comma separated variable names that may be used inside the function body.
Calling a function with arguments is done by separating the arguments with commas as well.
Functions may have as many arguments as required.

```
def <name>(<arg0>, <arg1>, ...):
    <body>
```

Functions

Arguments

As an example, a function that prints either 42 or zero, depending on the value of a single argument `x`.

```
def forty_two_or_zero(x):
    if x:
        print(42)
    else:
        print(0)

# call the function
forty_two_or_zero(True)
# prints 42

# call with another argument
dont = False
forty_two_or_zero(dont)
# prints 0
```

Functions

Arguments

To demonstrate multiple arguments, let us reimplement the power function.

Note that the order of the arguments in the function definition is important. Python calls them **positional arguments**.

```
def power(base, x):
    return base**x
```

Functions

Calling other functions

Functions may of course call other function in their body. Let us look at the `sin(1/x)` example again.

```
from math import sin

def sin_inv_x(x):
    if x == 0.0:
        result = 0.0
    else:
        result = sin(1.0/x)
    return result
```

Functions

Docstrings

Functions may have optional documentation embedded within them. Documenting your code is always a good idea. Python allow to describe what a function does using a string literal as the first thing in a function body.

```
def <name>(<args>):
    """<docstring>"""
    <body>
```

The docstring should be descriptive but concise. They can be displayed with Python's built-in `help()` function.

```
def power(base, x):
    """Computes base^x. Both base and x should be integers
    floats, or other numeric types
    """
    return base**x
```

DEMO

Functions

Keyword Arguments

Default values for arguments are a feature for when arguments should have a standard behavior. In Python these are known as **keyword arguments** and have three advantages.

- Keyword arguments are optionally supplied when the functions is called - reducing what must be explicitly passed
- When used by name they may be called in any order
- help define the kinds of values that may be passed into the function

The syntax is as follows:

```
def <name>(<arg0>, <arg1>, ..., <kwarg0>=<val0>,
           <kwarg1>=<val1>, ...):
    '''docstring'''
    <body>
```

Functions

Keyword Arguments

As an example let us consider the polynomial `ax+b`.
This should be implemented as a function with `a` and `b` having the default values of 1 and 0, respectively.

```
def line(x, a=1.0, b=0.0):
    return a*x + b
```

The `line()` function can now be called with neither `a` nor `b`, either `a` or `b` or both as parameter.
The **positional argument** `x`, however, must be given at any function call.

Functions

Keyword Arguments

Here are a few exemplary calls to the `line()` function.

```
line(42)              # no keyword args, return 1*42 + 0
line(42, 2)           # a=2, returns 84
line(42, b=10)        # b=10, return 52
line(42, b=10, a=2)   # returns 94
line(42, a=2, b=10)   # also returns 94
```

Functions

Mutable data types such as lists, sets, and dicts should *never* be used as default values.
Because they retain their state from one function call to the next.
This can lead to very weird and hard to find bugs.

```
# never ever do this
def myappend(x, liste=[]):
    liste.append(x)
    print(liste)
    return liste

myappend(6)              # [6], seems right
myappend(42)             # [6, 42], should be [42]
myappend(12, [1, 16]) # [1, 16, 12], hmm ok
myappend(65)             # [6, 42, 65], what the??
```

Functions

If you really have to do something like that you can use the `None` keyword.

```
def myappend(x, liste=None):
    if liste is None:
        liste = []
    liste.append(x)
    print(liste)
    return liste

myappend(6)             # [6]
myappend(42)            # [42]
myappend(12, [1, 16]) # [1, 16, 12]
myappend(65)            # [65]
```

Functions

Variable number of arguments

Some functions may take a variable number of arguments.
To see why this is useful, let us look at Python's built-in `max()` function.
This function will take any number of arguments and always return the highest value.

```
max(6, 2)           # 6
max(6, 42)          # 42
max(1, 16, 12)      # 16
max(65, 42, 2, 8) # 65
```

To write functions that take a variable number of arguments, the function definition must contain a single special argument that may have any name but is prefixed with an asterisk (*).

```
def <name>(<arg0>, <arg1>, ...,
           <kwarg0>=<val0>, <kwarg1>=<val1>, ...,
           *<args>):
    '''docstring'''
    <body>
```

Functions

Variable number of arguments

Let us write our own version of a minimum function:

```
def minimum(*args):
    '''Takes any number of arguments
       and returns the lowest value
    '''
    m = args[0]
    for x in args[1:]:
        if x < m:
            m = x
    return m
```

`args` is here a tuple (immutable list) which contains all the arguments.

Functions

Variable number of arguments

Our minimum function can be called as the `max()` function before. However since `args` is a tuple we can also unpack an existing sequence into it when calling the function. This can be done using the same asterisk notation.

```
minimum(6, 42)          #6

data = [65, 42, 2, 8]
minimum(*data)          # 2
```

This means you can prepare the data you want to send to a function beforehand.

Functions

Variable number of keyword arguments

Of course this also works with keyword arguments. The supplied keyword arguments are put in a dictionary. This time the syntax uses a double asterisk (**).

The final syntax for a function definition thus becomes:

```
def <name>(<arg0>,...,<kwarg0>=<val0>,...,*<args>,**<kwargs>):
      '''docstring'''
      <body>
```

Functions

Multiple return values

In Python, as with many other languages, only one *object* can be returned from a function. However, the packing and unpacking semantics of tuples mimic the behavior of multiple return values. The statement `return x, y, z` may look like 3 return values. In reality a 3-tuple is created and that tuple is returned. This tuple can then either be unpacked or remain as a tuple.

Functions

Multiple return values

Let us consider a function that takes mass and velocity as arguments and returns momentum and energy.

```
def momentum_energy(m, v):
    p = m * v
    e = 0.5 * m * v**2
    return p, e
```

This function my be called in either of the following ways:

```
# return a tuple
p_e = momentum_energy(42.0, 65.0)
print(p_e)
# (2730.0, 88725.0)

# unpack the returned tuple
mom, eng = momentum_energy(42.0, 65.0)
print(mom)
# 2730.0
```

Functions

Scope

Function scope is a very important concept necessary to understand how functions *work* and how they enable reuse of code. All functions share the notion that variable defined inside of the function have lifetimes that end when the function returns. This is known as **local scope**. When the functions returns, all local variables "go out of scope" and their resources may be safely removed.

Variables defined outside the function have **global scope** with respect to the function at hand. Global variables may be accessed and modified if their names are not overridden by a local variable with the same name.

Global scope is also sometimes called **module scope** because variables at this level are global only to the module (the `.py` file) where they live.

Functions

Scope

In the following example the variables `a`, `b`, and `c` are in the global scope and the variables `x`, `y`, and `z` are local to `func()`.

```
# global scope
a = 6
b = 42

def func(x, y):
    # local scope
    z = 16
    return a*x + b*y + z

# global scope
c = func(1, 5)
```

Functions

Scope and nested functions

Functions may be nested in other functions. Inner functions then share the scope with the outer function, but not the other way.

```
# global scope
a = 6
b = 42

def outer(m, n):
    # outer's scope
    p = 10

    def inner(x, y):
        # inner's scope
        return a*p*x + b*n*y + z

    # outer's scope
    return inner(m+1, n+1)

# global scope
c = outer(1, 5)
```

Functions

Scope, Overriding Variables

Suppose a function assigns a variable to a name that already exists in global scope. The global value is overridden for the remainder of the function call. The global variable remains unchanged.

```
a = 6

def a_global():
    print(a)

def a_local():
    a = 42
    print(a)

a_global()   # 6
a_local()    # 42
print(a)     # 6
```

Functions

Scope, Overriding Variables

But beware! You cannot mix the usage of global and local variables of the same name inside of a function.

```
a = 'A'

def func():
    # you cannot use the global 'a' because ...
    print('Big ' + a)
    # a local 'a' is eventually defined!
    a = 'a'
    print('small ' + a)

func()
```

The above defined function `func()` will throw an `UnboundLocalError` exception.

Functions

Scope, Overriding Variables

However, there is a solution: the `global` keyword.
You can explicitly say that a variable used inside a function lives in global scope.

```
a = 'A'

def func():
    global a
    print('Big ' + a)
    a = 'a'
    print('small ' + a)

func()                    # Big A
                          # small a
print('global + ' a) # global a
```

Functions

Recursion

In order to understand recursion, you must first understand recursion.

Since a function name is part of the surrounding scope, a function has access to its own name from within its own body. That means a function can call itself. This is known as recursion.

The simplest bad example is the following:

```
# DO NOT RUN THIS
def func():
    func()
```

Functions

Recursion

The classical example for recursion though, is the calculation of the `n`-th value of the Fibonacci sequence.

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Here, for all cases where `n>1`, the `fib()` function is called for `n-1` and `n-2`. However, zero and one are cases for which further calls to `fib()` do not occur. This recursion terminating property make zero and one **fixed points** of the Fibonacci function. More mathematically, fixed points are defined such that `x` is a fixed point of a function `f()` if and only if `x == f(x)`.

Functions

Lambdas

Lambdas are a special way of creating small, single line functions. Unlike normal functions, lambdas are an expression rather than a statement. There are certain restrictions when dealing with lambdas.

- must compute only a *single* expression
- no statements allowed
- cannot assign local variables
- evaluation of expression is *always* returned

The syntax is as follows:

```
lambda <args>: <expr>
```

Functions

Lambdas

Some examples for lambdas in Python.

```
# a simple lambda
lambda x: x**2

# a lambda that is called after it is defined
(lambda x, y=10:2*x + y)(42)

# we can also name a lambda
f = lambda: [x**2 for x in range(10)]
f()

# lambda as keyword argument in a function call
func([6, 26, 496, 8128], lambda data: sum([x**2 for x in data]))
```

Functions

Lambdas

Historically, lambdas come from the **lambda calculus** which helps for the mathematical basis for computation. This topic has spawned its own language paradigm called **functional programming**. Unlike in object oriented programming, where everything is an object, in functional languages everything is a function. Functional languages such as Lisp, Haskell, and OCaml have been gaining popularity recently.