732A74 lecture 1

- Intro
- What Python is
- Development environments
- Finding information
- Data types
- Control structures
- General-purpose hints

Course intro

- Python is everywhere
- Often batteries included
- ...and a large ecosystem
- Multiparadigm[ish]. Functional, object-oriented...
- The role of this course. See Course webpage (https://www.ida.liu.se/~732A74/)

Course structure

- Labs (subdivided)
  - Lab 1 - Basic Python
  - Lab 2 - Functions, some debugging, functional patterns
  - Lab 3 - Objects, OOP and using Python
- Pass/fail
- Previous course evaluations
- What is expected of you
  - Pair up! Webreg (https://www.ida.liu.se/webreg3/732A74-2019-1/LABA)
  - Follow the lab rules!
  - Self-study.
  - Expect to look for information yourself.
- Presentation by Anders Märak Leffler
- Attribution: extends work by Johan Falkenjack.
- License: CC-BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0/)

Philosophy

- About Python, not data science.
- Language course, with pointers to useful issues.
- Some useful general-purpose tools (and programming patterns).
- *Not* a course on proper software engineering, testing, computer science...
  - Need more basic programming help? Ask. We might be able to point to other materials.

Python development environments

- Python REPL and IPython
- Any text editor and python3 interpreter.
- IDLE
- General IDE:s (PyCharm, Eclipse, Visual Studio etc)
- Scientific IDE:s (Spyder, Rodeo etc)
- Notebooks (like the ones in this course)

Getting Python

- Full distributions
  - CPython (the standard). Download on python.org (python.org).
  - Anaconda
  - PyPy
  - Jython
- Package Managers
  - pip
  - conda

What is Python?

- "New" and "old" language, first out 1991
- Created and directed by self-proclaimed "Benevolent Dictator for Life" Guido van Rossum (ex-BDFL (https://mail.python.org/pipermail/python-committers/2018-July/005664.html))
- High-level language
- Not "close to the metal" by default
- ...but libraries can help.
- Useful **glue** between programs (eg C/C++).
- Emphasizes readability

In [1]:

```
# Ex: calculating the average of (non-empty) sequence.
# this is a comment. It starts with #
def average(seq):
    return sum(seq) / len(seq)
average([1,2,3,4])
```

Out[1]:

```
2.5
```

Ex #2: Text mining intro (https://www.ida.liu.se/~732A47/info/courseinfo.en.shtml)

- Note: two currently developed versions, 2.x and 3.x. This course uses Python 3.
  - Telltale sign of python2: print "hello" instead of print("hello").
  - xrange instead of range. (Though there is a range in Python 2 as well...)

Python in comparison

- Not primarily a numerical language, no built-in vectors, matrices (with efficient implementation of operations) or the like.
- Interpreted/JIT-compiled [compilation with Cython optional]
- Automatic memory management. (Cf Java, Racket, rather than C/C++).
- Strongly typed (like Haskell, C++).

In [2]:

```
# Lists are not maths vectors!
[1,2] * 5
```

Out[2]:

```
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

In [4]:

```
# Strongly typed

99 + "Luftballon" # Should crash. number + string not ok!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-aaca5ac8fb9c> in <module>()
      1 ##### Strongly typed
      2
----> 3 99 + "Luftballon"

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

- Dynamically typed (checked runtime, can change)

In [6]:

```
# Dynamically typed.
mystr = "hello"
mystr
```

Out[6]:

```
'hello'
```

In [7]:

```
mystr = 5
mystr
```

Out[7]:

```
5
```

In [10]:

```
# Does "a" have a type that changes?
a = 99
type(a)
```

Out[10]:

```
int
```

In [11]:

```
a = "asdasdasd"
type(a)
```

Out[11]:

```
str
```

In [ ]:

```
# Seems like it...
"""
 But: in python, a is just a label. The _value_
 (99 or the string) has the type.
"""
```

- In general: we'll do it live! Errors when you run, rather than when you write.
  - Test your code before you run it overnight (or ship it).
- Style: duck typing (http://blog.helloruby.com/post/70507494778/day-19-duck-walk-one-day-ruby-walks-in-the-forest). This is used for polymorphous behaviour. [Caveat: from book about Ruby]

Peculiarities

- Indentation means something. Groups code together. (Cf { ... }, or Haskell).

In [13]:

```
# Function def.
def f():
    print("this is in the function body")
```

In [14]:

```
def g():
    print("asdasdasd")
print("Outside the function")
```

```
Outside the function
```

In [16]:

```
g()
```

```
asdasdasd
```

In [17]:

```
def h():
    print("Inside!")
print(12125345)
    print("Let's try to get back into the function.")
```

```
  File "<ipython-input-17-a1e2916faf91>", line 4
    print("Let's try to get back into the function.")
    ^
IndentationError: unexpected indent
```

- Multiple simultaneous assignments.

In [23]:

```
a = b = 100
```

In [21]:

```
b
```

Out[21]:

```
100
```

In [22]:

```
# Assignments do not have a value of themselves.
# (Like in some other languages)
print("The value of the assignment is ", a = 5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-22-15dba5031807> in <module>()
      1 # Assignments do not have a value of themselves.
      2 # (Like in some other languages)
----> 3 print("The value of the assignment is ", a = 5)

TypeError: 'a' is an invalid keyword argument for print()
```

- Everything is an object, including modules. Special import syntax.

In [25]:

```
# Getting sqrt
import math
```

In [26]:

```
math.sqrt(100)
```

Out[26]:

10.0

In [28]:

```
# Getting help
help(math)
```

```
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x.

    asinh(x, /)
```

In [29]:

```
# Get all the exposed members
dir(math)
```

Out[29]:

```
['__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
 'atan',
 'atan2',
 'atanh',
 'ceil',
 'copysign',
 'cos',
 'cosh',
 'degrees',
 'e',
 'erf',
 'erfc',
 'exp',
 'expm1',
 'fabs',
 'factorial',
 'floor',
 'fmod',
 'frexp',
 'fsum',
 'gamma',
 'gcd',
 'hypot',
 'inf',
 'isclose',
 'isfinite',
 'isinf',
 'isnan',
 'ldexp',
 'lgamma',
 'log',
 'log10',
 'log1p',
 'log2',
 'modf',
 'nan',
 'pi',
 'pow',
 'radians',
 'remainder',
 'sin',
 'sinh',
 'sqrt',
 'tan',
 'tanh',
```

```
 'tau',
 'trunc']
```

In [31]:

```
# Importing cos, pow specifically
from math import cos, pow
pow(2,999)  # note: no math.<sth>
```

Out[31]:

```
5.357543035931337e+300
```

In [32]:

```
# Long module names can be abbreviated upon import
import math as m
m.sin(0)
```

Out[32]:

```
0.0
```

Note (mostly outside this course): When you create a separate file, it automatically becomes a module. A file can import and export bindings in its namespace (a large package doesn't need to be written all in one file). See the documentation.

Python objects and types

- No primitive types.
- Plenty of builtins: **string**, **int**, **list**,...

In [ ]:

```

```

A brief note on numbers

- At a high level: works as you might expect.

In [34]:

```
# An integer
5 + 3
```

Out[34]:

```
8
```

In [6]:

```
a = 5
type(a)
```

Out[6]:

```
int
```

In [3]:

```
# A float
b = 123.923
b
type(b)
```

Out[3]:

```
float
```

In [7]:

```
type(a + b)
```

Out[7]:

```
float
```

- Common ancestors, and can usually be converted. Inheritance in a later lecture.

In [9]:

```
int(b) # remember: b was bound to a float value.
c = 0.9
int(c)
```

Out[9]:

```
0
```

- The expected operations. Comparison using == (as with many other types).

In [41]:

```
5 == 123
```

Out[41]:

```
False
```

In [42]:

```
5 == 5.0
```

Out[42]:

```
True
```

- Size handled automatically for *standard builtin types*.

In [13]:

```
4**3
```

Out[13]:

64

Caveat: If you need to worry about overflows, they are a headache beyond you as a novice Python programmer. Or it will likely be using special types where this is mentioned in the module help.

- A deeper consequence: "similar" numbers may or may not be the same object. Practically: use `==` rather than `is` to test if numbers are the same.

In [46]:

```
5 == 5
```

Out[46]:

True

In [47]:

```
5 is 5  # are they the same object?
```

Out[47]:

True

In [49]:

```
a = 2**50
b = 2**50
a is b
```

Out[49]:

False

Use == !

Strings

- There are **no characters**, only strings (possibly of length one).
- Immutable (meaning?)
- 'some' creates a string here, as does "thing", """possibly multiline"""

In [52]:

```
type('a')
```

Out[52]:

```
str
```

In [55]:

```
mystr = "Hello world"
mystr
```

Out[55]:

```
'Hello world'
```

In [57]:

```
mystr = 'Hello "word".'
mystr
```

Out[57]:

```
'Hello "word".'
```

In [59]:

```
mystr = """This has many lines
asdadasd"""
print(mystr)
```

```
This has many lines
asdadasd
```

- Python is a language for text processing. Plenty of useful methods!

...but how do we find them?

In [84]:

```
animals = "Snakes"
```

In [61]:

```
dir(animals)
```

Out[61]:

```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmod__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'capitalize',
 'casefold',
 'center',
 'count',
 'encode',
 'endswith',
 'expandtabs',
 'find',
 'format',
 'format_map',
 'index',
 'isalnum',
 'isalpha',
 'isascii',
 'isdecimal',
 'isdigit',
 'isidentifier',
 'islower',
 'isnumeric',
 'isprintable',
 'isspace',
 'istitle',
```

```
 'isupper',
 'join',
 'ljust',
 'lower',
 'lstrip',
 'maketrans',
 'partition',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

In [64]:

```
animals.lower()
```

Out[64]:

```
'snakes'
```

In [65]:

```
"hello world".title()
```

Out[65]:

```
'Hello World'
```

- Indexing. Starts before the first character.

In [68]:

```
# Indexing.
animals[0]
```

Out[68]:

```
'S'
```

In [70]:

```
# Negative indices.
animals[-2]
```

Out[70]:

'e'

In [71]:

```
animals[999] # Should crash!
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-71-1e0b4af99a29> in <module>()
----> 1 animals[999]

IndexError: string index out of range
```

- Slicing. [start:end] or [start:end:step]

In [72]:

```
animals[1:4]
```

Out[72]:

'nak'

In [73]:

```
animals[4:1:-1]
```

Out[73]:

'eka'

In [74]:

```
animals[:3]
```

Out[74]:

'Sna'

In [75]:

```
animals[1:]
```

Out[75]:

'nakes'

In [76]:

```
animals[:-1]
```

Out[76]:

'Snake'

- Repeating patterns.

In [78]:

```
(animals + " ")
```

Out[78]:

'Snakes '

In [79]:

```
#(animals + " ")*500
```

Out[79]:

```
'Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes
Snakes Snakes Snakes Snakes Snakes Snakes Snakes Snakes '
```

- Concatenation by + (note: does not scale well!)

In [85]:

```
presentation = "Great " + animals
presentation
```

Out[85]:

'Great Snakes'

In [86]:

```
# What will this yield? Error? OK?
presentation += " is something that captain Haddock of Tintin often exclaims."
presentation
```

Out[86]:

'Great Snakes is something that captain Haddock of Tintin often exclaims.'

- Breaking up strings (very useful!).

In [89]:

```
pres_words = presentation.split()
pres_words
```

Out[89]:

```
['Great',
 'Snakes',
 'is',
 'something',
 'that',
 'captain',
 'Haddock',
 'of',
 'Tintin',
 'often',
 'exclaims.']
```

In [90]:

```
help(presentation.split)
```

```
Help on built-in function split:

split(sep=None, maxsplit=-1) method of builtins.str instance
    Return a list of the words in the string, using sep as the delimiter string.

    sep
      The delimiter according which to split the string.
      None (the default value) means split according to any whitespace,
      and discard empty strings from the result.
    maxsplit
      Maximum number of splits to do.
      -1 (the default value) means no limit.
```

- String formatting. Often useful for recurrent string injections.

In [91]:

```
"{0}, {0}, {1} {0}".format(animals, "several")
```

Out[91]:

'Snakes, Snakes, several Snakes'

In [94]:

```
# The pattern is itself a (reusable) string.
pattern = "{0:.3}, {1}"
pattern.format(3.7198234234234234234112312323, animals)   # lots of decimals
# How do we find this out (if we need it)?
# check docs.python.org (and make sure that it's the right version)
```

Out[94]:

'3.72, Snakes'

- Joining together strings.

In [100]:

```
words = presentation.split() # Divide the words that we should join.
" ".join(words)
```

Out[100]:

'Great Snakes is something that captain Haddock of Tintin often exclaims.'

Note: much faster than repeat concatenation.

In [101]:

```
# Bonus
import profile

# All loops unrolled

N = 999999      # Don't go over 99999

print("Generating and loading concatenation code.")
conc = "def concat_test():\n"
conc += '  mystring = ""\n'
conc += '  mystring += "NaNaNaNa"\n'*N
exec(conc)  # Execute the code as Python.

print("Generating and loading join code.")
joint = "def join_test():\n"
joint += '  "".join({})\n'.format(("NaNaNaNa "*N).split())
exec(joint)

print("Generating and executing code")
print("--- String concatenation")
profile.run("concat_test()")
print("--- Using join")
profile.run("join_test()")
```

```
Generating and loading concatenation code.
Generating and loading join code.
Generating and executing code
--- String concatenation
         5 function calls in 0.079 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.079    0.079 :0(exec)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.079    0.079 <string>:1(<module>)
        1    0.079    0.079    0.079    0.079 <string>:1(concat_test)
        1    0.000    0.000    0.079    0.079 profile:0(concat_test())
        0    0.000             0.000          profile:0(profiler)


--- Using join
         6 function calls in 0.019 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.019    0.019 :0(exec)
        1    0.013    0.013    0.013    0.013 :0(join)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.019    0.019 <string>:1(<module>)
        1    0.006    0.006    0.019    0.019 <string>:1(join_test)
        1    0.000    0.000    0.019    0.019 profile:0(join_test())
        0    0.000             0.000          profile:0(profiler)
```

In [ ]:

```
# Bonus material (_not_ required!)
# If we have lots of extra time, set N = 3, load the code above and consider the code generated.
# How does it look? Memory accesses?
import dis      # Disassembly module. Look at the Python bytecode.
# dis.dis(concat_test)
# dis.dis(join_test)
```

- All objects will have a string representation (probably useful). Remember to use this when concatenating.

In [103]:

```
99 + "Luftballon"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-103-fd639336754b> in <module>()
----> 1 99 + "Luftballon"

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In [105]:

```
str(99) + " Luftballon"
```

Out[105]:

```
'99 Luftballon'
```

Lists

- Lists are ordered containers of **several** values, **possibly of different types**.

In [106]:

```
seq_a = [1,2]
my_seq = ["Snakes", "Snakes", "several", "Snakes", 100, seq_a, 99]
```

- Lists are indexable and slicable (like strings).

In [110]:

```
seq_a[0]
```

Out[110]:

```
1
```

- Lists are **mutable**.

In [113]:

```
seq_a.append(99923423423499)
seq_a
```

Out[113]:

```
[1, 2, 99999, 99999, 99923423423499]
```

- Elements are *not* named, and lists are *not vectors* in a mathematical sense.
- + will concatenate, and create a new list (like with strings).

In [115]:

```
seq_a + [123]
```

Out[115]:

```
[1, 2, 99999, 99999, 99923423423499, 123]
```

In [116]:

```
seq_a #unchanged
```

Out[116]:

```
[1, 2, 99999, 99999, 99923423423499]
```

- There are several useful methods. Extra noteworthy: `append`, `count`.

In [ ]:

```
# left
```

- Like many containers (and strings!) they support membership testing using `in`.

In [119]:

```
"my string" in seq_a
```

Out[119]:

```
False
```

In [121]:

```
"Snakes" in my_seq
```

Out[121]:

```
True
```

- We many have nested lists.

In [124]:

```
names = ["Alonzo", "Zeno"]
seq = [1,2, names, ["Alonzo", "Zeno"]]
```

In [126]:

```
seq[-1] # the last value is a list
```

Out[126]:

```
['Alonzo', 'Zeno']
```

- Indexing will produce an element (of the list). Slicing will always produce a list! (In strings it was always ever a string.)

In [127]:

```
names[1]
names[0:1]
```

Out[127]:

```
['Alonzo']
```

- **Lists being mutable has consequences**. Shared structures.

In [128]:

```
names = ["Alonzo", "Zeno"]
names_2 = ["Alonzo", "Zeno"]
big_list = [1,2, names, names_2]
names[0] = 999999
# What will big_list be?
big_list
```

Out[128]:

```
[1, 2, [999999, 'Zeno'], ['Alonzo', 'Zeno']]
```

Note how two lists may have the same elements, but still be different lists. (More on this later.)

- Subtle difference between `seq = seq + some_list` and `seq += some_list`.

In [ ]:

```
# left
```

Booleans and their operators

- `True` and `False`
- `and`, `or`, `not`

In [129]:

```
a = True
b = False
a or b
```

Out[129]:

```
True
```

In [130]:

```
a and b
```

Out[130]:

```
False
```

In [131]:

```
# Noteworthy "weird" uses cases

a and "something else"
```

Out[131]:

```
'something else'
```

Dictionaries

- Keystone of practical Python.
- Unordered collections of pairs.
  - Often key-value mappings.
  - Sometimes used for sparse data.
- Arbitary types of values. Keys must be hashable (approximately immutable).
- Based on hash tables. Fast lookup of *keys*.

In [132]:

```
words = { "hej" : 3, "hopp" : 4,  "thesaurus" : 9 }
words["hej"]
```

Out[132]:

```
3
```

In [133]:

```
"hej" in words
```

Out[133]:

```
True
```

In [136]:

```
words.get("some key that isn't there",999)
```

Out[136]:

```
999
```

- Dictionaries are iterable.

In [137]:

```
for key in words:
    print(key)
```

```
hej
hopp
thesaurus
```

- Dictionaries have useful methods. In particular, `items`.

In [ ]:

```

```

Tuples

- Tuples are **immutable** sequences of arbitrary values.
- They support most of the methods that lists support.
- Can be used as keys in dictionaries.
- Handle multiple return values from functions.
- **If you don't need mutability, prefer tuples to lists**. A lot less copying (and smaller memory footprint).

In [140]:

```
import sys
sys.getsizeof( (1,2,3,4,5,6,7) )  # Tuple
sys.getsizeof( [1,2,3,4,5,6,7] )  # List case
```

Out[140]:

```
120
```

A note on type conversions and Pythonic ways

- The standard builtin collection constructors support iterables. More of this in a later lecture. Plainly: you can convert back and forth between them easily.

In [142]:

```
seq = [1,2,3]
pairs = [ ("key1",2), ("key2",4) ]
tuple(seq)
dict(pairs)
```

Out[142]:

```
{'key1': 2, 'key2': 4}
```

- We can *unpack* tuples, lists and other iterable values:

In [1]:

```
first, second = (1, 2)
first
```

Out[1]:

```
1
```

In [20]:

```
[foo, bar] = (9,2)
bar
a=(1,2)
print("the elements are  {0},{1} :".format(a[0],a[1]))
```

```
the elements are  1,2 :
```

In [145]:

```
a,b,c,d = [1,2,3 ]    # bad number of values
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-145-46e9697148b3> in <module>()
----> 1 a,b,c,d = [1,2,3 ]    # bad number of values

ValueError: not enough values to unpack (expected 4, got 3)
```

Side note for language geeks: this is not the kind of general pattern matching that we find in Haskell or Erlang.

Control structures

Conditionals - if

- Python supports if-then-else, with elif.

In [146]:

```
val = int(input("Enter a number: "))
if val > 9000:
    print("Big")
elif val > 8000:
    print("Moderate")
elif val > 0:
    print("Meh")
else:
    print("Tiny")
```

```
Enter a number: 5
Meh
```

- In `if` statements, we don't need to be exhaustive.

In [147]:

```
val = int(input("Enter a number:"))
if val < 0:
    print("Warning! Abort! Abort!")
    # Possibly break execution here
```

```
Enter a number:112
```

In [ ]:

```

```

- `if` uses "truthiness", not True/False. This might be a cause for confusion.

In [148]:

```
val = "Sir Michael Palin"
if val:
    print(val, "KCMG, CBE, FRGS")
else:
    print("Someone else.")
```

```
Sir Michael Palin KCMG, CBE, FRGS
```

- Some "falsey" values are [], "", 0, {} (by convention: the "empty" value of any type). The Pythonic way is to use this to write polymorphic code!

In [152]:

```
val = [1,2,3]
if not val:
    print("The sequence is empty.")
else:
    print("There is a there there.")
```

```
There is a there there.
```

- "False friends": There is a similar-looking `if` expression. This is an expression (rather than something to control flow). Thus it should *always* be possible to replace it with a value. We always require both *then* and *else*.

In [ ]:

```
val = input("What is thy quest? ")
reply = "That's great!" if val == "python" else "Why?"
print(reply)
```

- If we really insist, we may use dictionaries as (or emulating part of the behaviour of) switch statements.

while loop

In [153]:

```
val = 5
while val > 0:
    val = val - 1
    print(val, "bottles of beer on the wall.")

print("All done.")
```

```
4 bottles of beer on the wall.
3 bottles of beer on the wall.
2 bottles of beer on the wall.
1 bottles of beer on the wall.
0 bottles of beer on the wall.
All done.
```

- We can break using `break` and continue using `continue`.

In [154]:

```
val = 5
while val > 0:
    val = val - 1
    continue  # Skip the rest of the loop body.
    print(val, "bottles of beer on the wall.")
print("All done.")
```

```
All done.
```

- A Python feature is while-else. Interpret the `else` as "no break occurred".

In [155]:

```
val = 5
while val > 0:
    val = val - 1
    print(val, " bottles of beer on the wall.")

else:
    print("OK execution, no breaks.")
```

```
4  bottles of beer on the wall.
3  bottles of beer on the wall.
2  bottles of beer on the wall.
1  bottles of beer on the wall.
0  bottles of beer on the wall.
OK execution, no breaks.
```

The misnamed for loop

- We can use `for` to iterate over values. Below, we use a `range(0)` expression to get numbers 0,..,n-1.

In [156]:

```
for i in range(4):
    print("The current value is {}".format(i))
```

```
The current value is 0
The current value is 1
The current value is 2
The current value is 3
```

- for _ in _ should be read as "for each _ in [something iterable], do the following".
- Can be used to iterate over iterables. lists, strings, dictionaries...

In [157]:

```
# Note the naming of the loop variable.

for name in ["Alonzo", "Zeno"]:
    print("{} is a happy cat".format(name))
```

```
Alonzo is a happy cat
Zeno is a happy cat
```

- We can unpack values directly in the loop.

In [158]:

```
scores = {"UK" : 5, "Germany" : 2, "Sweden" : 1}
for key, value in scores.items():
    print("{} got {}".format(key,value))
```

```
UK got 5
Germany got 2
Sweden got 1
```

In [ ]:

```
# Conundrum I: infinite loop?
for i in range(4):
    print("The current value is {}".format(i))
    i = 0
```

In [ ]:

```
# Conundrum II: what will this print?
i = "Hello"
print("Before, i is", i)
for i in range(4):
    print(i)
print("Afterwards, i is", i)
```

- `for` loops support `break`, `continue` and also have an `else`.

Comprehensions

- Powerful feature, easy to read. Efficient. Use them!
- In mathematics: $\{f(x)|x \in A\}$

Ex: The list of x^2 for all $x \in \{0, 1, \ldots, 9\}$.

In [159]:

```
[x*x for x in range(10)]
```

Out[159]:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Ex: The list of x^2 for all $x \in \{0, 1, \ldots, 9\}$ where x is divisible by three.

In [160]:

```
[x*x for x in range(10) if x % 3 == 0]
```

Out[160]:

```
[0, 9, 36, 81]
```

- There are dictionary comprehensions.

In [161]:

```
{ name : score for name, score in [("UK", 5), ("Peru", 99)]}
```

Out[161]:

```
{'UK': 5, 'Peru': 99}
```

- Philosophical (and useful) note: the comprehension expression *itself* produces a value.

In [163]:

```
some_squares = ( (x, x*x) for x in range(10) )
print(some_squares)
constructor = dict
print("After we use " +  str(constructor) + " we get", constructor(some_squares))
```

```
<generator object <genexpr> at 0x7f0c283bc138>
After we use <class 'dict'> we get {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7:
49, 8: 64, 9: 81}
```

In [ ]:

```
# We can iterate over the values immediately.
for i, i_sq in ( (x, x*x) for x in range(10) ):
    print(i, i_sq)
```

We will return to the notion of generators later.

- A note on efficiency.

In [165]:

```
import profile


def loop_test(N):
    vals = []
    for i in range(N):
        vals.append(N*N)
    return vals

def comprehension_test(N):
    return [i*i for i in range(N)]

N = 999999

print("---- Testing standard for loop")
dummy = profile.run("loop_test({})".format(N))

print("---- Testing comprehension")
dummy = profile.run("comprehension_test({})".format(N))
```

```
---- Testing standard for loop
         1000004 function calls in 1.898 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   999999    0.894    0.000    0.894    0.000 :0(append)
        1    0.000    0.000    1.898    1.898 :0(exec)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.995    0.995    1.889    1.889 <ipython-input-165-f91e71777d10>:4(loo
p_test)
        1    0.009    0.009    1.898    1.898 <string>:1(<module>)
        1    0.000    0.000    1.898    1.898 profile:0(loop_test(999999))
        0    0.000             0.000          profile:0(profiler)


---- Testing comprehension
         6 function calls in 0.068 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.068    0.068 :0(exec)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.059    0.059 <ipython-input-165-f91e71777d10>:10(co
mprehension_test)
        1    0.059    0.059    0.059    0.059 <ipython-input-165-f91e71777d10>:11(<l
istcomp>)
        1    0.009    0.009    0.068    0.068 <string>:1(<module>)
        1    0.000    0.000    0.068    0.068 profile:0(comprehension_test(999999))
        0    0.000             0.000          profile:0(profiler)
```

Note: if you are super interested in this, check out the disassembly of the loop_test function and consider all the conditional jumps, memory accesses and and list resizing.

Conclusion for everyone else: prefer comprehensions unless you have reason not to.

Files

- Read about them in the documentation.

General hints

- The documentation (https://docs.python.org) is helpful.
- The Python tutorial (https://docs.python.org/3/tutorial/index.html) can be useful.

In [ ]:

```
# How does the append method of the list seq work?
seq = [1,2,3]
```

In [ ]:

```
# Which methods does seq support anyway? [notebook]
```

In [ ]:

```
# Which methods does seq support anyway? [in general]
```

- Google!
- Common source of errors: we use **Python 3**, some sites will provide Python 2.x code.
- Be each others' ducks (http://blog.helloruby.com/post/70582154912/day-20-talk-to-the-duck-whenever-ruby-runs-into)
- Ask the teachers!