Fundamentals of Programming for Digital Health

Part 2: Python

Berry Boessenkool use freely, cite 2023-01-27 11:08

Course content

- 5 weeks with lectures
- exercises in CodeOcean
  - can be solved in Rstudio / VScode
  - no tested support for other Python IDEs
- final exam (python only), Feb 09
- bonus points for improvements in slides / exercises / refCard

Python executive summary (source)

- interpreted language (no compilation)
- dynamic typing and binding (data type checked at run-time)
- object-oriented (data and functions)
- high-level (readable for humans)

Used in
data science, machine learning, web development, game gevelopment, robotics, autonomous vehicles, graphical user interface development, finance, ...

Resources

- download & install
- hints for Windows users
- tutorial
- standard libraries
- language reference + documentation
- Berry's RefCard (very close to course content)
- Other RefCard example + search
- towards data science translation guide R $<-->$ Python
- Computer Science Circles interactive Python course (much more verbose & detailed = slower than our course), German online course
- codingame.com: addictive problem solving coding competitions

Integrated development environments (IDE)

- PyCharm: Good for scientific development, but slow in startup
- VScode: (Visual studio code) increasingly popular, supports multiple languages, e.g. R
- RStudio: with integrated help + line-by-line execution!

- IDLE: Installed by default, not suitable for large projects
- programiz.com/python-programming/ide: Overview of more IDEs

- colab.research.google.com: Jupyter notebooks for Julia, Python, R
- Jupyter Manual

Code in these slides

```
# python code on grey boxes
```

```
# R code on green boxes
```

```
> shell/bash/cmd/terminal commands on yellow boxes
```

They are set manually, so please report mistakes :)

Exercises

If you want to use Rstudio: everything just like in the R part.

- Download exercise + run `codeoceanR::rt_create()` in R/Rstudio
- `CTRL` + `ENTER` or `CTRL` + `SHIFT` + `S`, `score.score()` at the bottom

For VScode:

- Download exercise
- Unzip manually
- Open all p**script*.py files in VScode
- Run line / selection / script, `score.score()` at the bottom

(Potential) first time settings:

- Manage restricted Mode - add Folder, click Trust
- Close Folder View
- `CTRL` + `,`, search 'execin', check Python>Terminal: execute in file dir
- Set keyboard shortcuts e.g. `Run Selection/Line` with `CTRL` + `ENTER`
- If `R is not found` on Windows, add it to the system (not user) PATH

```
x = "hello, python world!"
x
## 'hello, python world!'
```

In most circumstances (e.g. VScode, CodeOcean), an explicit print call is needed to generate any display:

```
print(x)
## hello, python world!
```

`print` is not needed in colab jupyter notebooks (create) and these slides (created with Rnw = R + LaTeX).

print

```
print("I'm writing:", x)
## I'm writing: hello, python world!
```

`print` concatenates multiple inputs separated by `sep` (default `" "`) and ending with `end` (default `"\n"`)

```
a = 5
print("a is:", a, 777, sep="0", end="\n\n\n")
print("a still is:", a)
## a is:050777
##
##
## a still is: 5
```

```
print("Berlin", end = "#")
print("Potsdam")
## Berlin#Potsdam
```

interactive user input

```
user_string = input()
print("The input was:", user_string)
user_string = input("Please write something here: ")
```

The output of `input` is always a character string, even if a number is given.

Operators, comments

```
5 + 8 # addition
## 13
6/7 # Spaces don't matter, like in R
## 0.8571428571428571
3 ** 2 # exponents (not with 3^2!)
## 9
```

```
19 // 3 # integer division
## 6
19 % 3 # modulus (remainder after dividing)
## 1
```

```
'''
this
is a
multi-line
comment
'''
```

shorthand assignment operator, errors

```
a = 17
a += 8 # not recommended for readability
a
## 25
```

```
a = 7
a *= a+2 # a = a*(a+2)
a
## 63
```

```
A + 88 # non-existing object, case SenSitIve
## NameError:  name 'A' is not defined
```

```
12b = 67 / 5 # object names cannot start with numbers
class = 4 # some statements not allowed as variable name
## SyntaxError:  invalid syntax
```

Extensive list of common errors, list of reserved keywords

SyntaxError: often forgotten brackets or colons (e.g. in loops)

```
from math import sqrt
```

the math module comes with python, but its functions are not loaded by default (unlike `print`, `+`, `input`, etc).

```
sqrt(700)
## 26.457513110645905
```

To explicitely attribute the source, use `module.function()`:

```
import math
math.log(55)
## 4.007333185232471
```

This renders code very readable and maintainable. If you use many functions from a single module, the first structure is preferred.

methods for objects

```
a_list_object = [42, 77, -5, 6] # c(42,77,-5,6) in R
a_list_object
## [42, 77, -5, 6]
```

```
a_list_object.append(111) # method for list objects
```

This changes the object without re-assignment (if mutable):

```
a_list_object
## [42, 77, -5, 6, 111]
```

If methods return an object, they can be chained:

```
char = "A Regular String"
```

```
char.count("r") # excludes upper case R
## 2
```

```
char.lower()
## 'a regular string'
```

```
char.lower().count("r")
## 3
```

Read lines of text file

```
fname = "textFile.txt"
with open(fname) as f:
    content = f.read()#.splitlines()
print(content)
## This is a little text file example
## to demonstrate reading lines into python.
## For real data, we'll use pandas in week 5.
```

`open("file.txt")` gives a connection with a .read method

`with` closes the connection (even in case of error)

`content.splitlines()` returns a list with 1 line per element

```
# in case absolute paths are ever needed:
import os, sys
os.getcwd() # directory at which the command is executed
sys.path # directory of main module (in development)
os.path.join(sys.path[0],"textFile.txt") # .py file path
```

Summary for 1.2 Syntax

syntax, objects, operators, functions:

- `obj = "string" # comment` ; `"""spans lines"""`
- `print("char", 42, obj, sep=" ", end="\n")`
- `user_string = input()`
- `+`, `-`, `*`, `/`, `**`, `//`, `%` and `+=`, `*=`
- `from module import function` ; `function(x)`
- `import module` ; `module.function(x)`
- `obj.method()`: `a_list.append`, `string.lower().count("p")`
- `with open("file.txt") as f:  content = f.read()`

Report unclear tasks in the forum.
Highlight the topics from this lesson in your RefCard.

Data types

```
type(5.67)
## <class 'float'>
```

```
type(5)
## <class 'int'>
```

```
type("char")
## <class 'str'>
```

```
print(7 > 4,  7 < 4)
## True False
type(7 > 4)
## <class 'bool'>
```

```
type(2+3j) # note the j instead of i
## <class 'complex'>
```

```
type(int("45"))
## <class 'int'>
```

```
isinstance(7.5, int) # check for class
## False
```

```
isinstance("Hello", (float, int, str) ) # one of types
## True
```

```
obj = "charstring"
print("Type is: ", type(obj) )
## Type is:  <class 'str'>
```

```
int(obj) # ValueError if object cannot be converted
## ValueError:  invalid literal for int() with base 10:
'charstring'
```

```
type(float("25"))
## <class 'float'>
```

type conversion demo

```
# Live demo final code:
f = float(input('Enter Temperature in °Fahrenheit: '))
c = (f-32) / 1.8
print("Temperature in °Celcius:", round(c, 1))
```

Function syntax, indenting

```
def greet(name, time="morning"):
    msg = "Hello, " + name + ". Good " + time + "!"
    return msg
```

```
greet('Bob', 'evening')
## 'Hello, Bob. Good evening!'
greet('Berry') # use the default argument (morning)
## 'Hello, Berry. Good morning!'
```

- colon `:` needed, indenting needed
- without explicit `return`, a function returns `None` (R: NULL)
- `return` exits function execution
- name & time are parameters, "Berry" & "evening" are arguments
- parameter=argument
- `print()` to the console ; `return` output that can be assigned
- IndentationError: wrong number of spaces at the beginning of a line

Function example

```
# load sqrt from built-in math module:
from math import sqrt

# pq-formula to find x values where y = x^2 + px + q is 0.
def pq(p,q):
    w = sqrt( p**2 / 4 - q )
    zeros = (-p/2 - w, -p/2 + w)
    return zeros

print(pq(3, -12))
## (-5.274917217635375, 2.274917217635375)
```

Lambda, scoping

```
multiply = lambda x,y: x*y # single-expression function
multiply(7, 3)             # on one line of code
## 21
```

```
def change_object():
    ab = 22
ab = 1
change_object()
ab # ab in global environment is still 1
## 1
```

```
def change_object():
    global ab     # Use only if you know what you're doing
    ab = 22
ab = 1
change_object()
ab # ab in global environment is now 22
## 22
```

assign several objects in a single line of code

```
def myfun(x, y):
    return x*2, y*2
```

```
a, b = myfun(3, 4)
a # two int objects, each with a single value
b
## 6
## 8
```

```
c = myfun(3, 4) # tuple object
c
## (6, 8)
```

swap two variables:

```
a, b = b, a # right hand side evaluated before assignment
```

built-in packages (modules)

We have already seen usage of some built-in packages:

```
from math import sqrt
import os, sys
import math
```

modules in the standard library $->$ no installation needed (list)

```
os.getcwd()
radius = float(input('enter radius: '))
print("perimeter is", 2 * math.pi * radius)
```

Good importing practice

3 ways to import code:

```
from math import *      # hard to attribute in longer code
```

```
from math import sqrt, pi    # elegant, but lots of typing
```

```
import math     # clear code: module.fun -> human friendly
math.pi
import math as m                  # less typing, still clear
m.pi
```

```
from random import random as r, randint as rint    # works
```

```
from random import random as r              # more readable
from random import randint as rint           # on two lines
```

external packages

source

Popular packages

- Data science: `numpy`, `pandas`
- Machine learning: `tensorflow`, `pytorch`
- Statistical analysis: `scipy`
- Web application: `django`
- Plotting: `matplotlib`, `seaborn`

install packages e.g. from pypi.org PYthonPackageIndex = CRAN for R
pip comes with python. In a terminal, write:

```
pip install numpy # pip3 on Mac, see next slide
pip list
```

Wheels makes installation fast.

Anaconda to install packages (also R) from their cloud.
In anaconda prompt: `conda install pandas ; conda list`

ImportError: wrong library/module name, non-existing objects

```
import pandas as pd
```

NOT FUN!!!

```
pip3 install numpy
pip3 list
```

Might not work either, try from R (they know how to do things):

```
install.packages("reticulate")
reticulate::install_miniconda()
reticulate::py_install("numpy")
```

```
from random import random, randint
random() # float between 0 (inclusive) and 1 ( exclusive)
randint(1, 6) # int between start and end, including them
```

Counting table

```
# sort(table(colors))[1:3] in R
colors = ['red', 'blue', 'blue', 'yellow', 'blue', 'red',
'green']
import collections
collections.Counter(colors).most_common(3)
## [('blue', 3), ('red', 2), ('yellow', 1)]
```

Read .py files

```
import os # built-in python module
os.getcwd().replace(os.sep, '/') # getwd() in R
## '/Users/berry/Dropbox/R/kurs/py_slides'
```

If at wd, there is *mydataset.py* with `age = 25`, we can use:

```
from mydataset import age
print(age+2)
## 27
```

Note: importing doesn't accept a filename (with .py extension).

```
from mydataset import * # to import all
source("mydataset.py") in R
```

```
import mydataset
mydataset.age
## 25
```

Objects defined in .py files

```
import mydataset
print("\n".join(dir(mydataset))) # objects in the module
## __builtins__
## __cached__
## __doc__
## __file__
## __loader__
## __name__
## __package__
## __spec__
## age
## job
## random
## simulate_job
```

Functions defined in .py files

```
type(simulate_job)
## <class 'function'>
```

```
import inspect
print(inspect.getfullargspec(simulate_job))
## FullArgSpec(args=[], ... defaults=None ...
```

```
simulate_job()
## 'programmer'
simulate_job()
## 'doctor'
```

```
print(inspect.getsource(simulate_job))
## def simulate_job():
##     jobs = ["teacher","plumber","doctor","programmer"]
##     return random.choice(jobs)
```

Character strings subsetting

```
b = "Hello, World!"
b[1] # 2nd letter
## 'e'
```

```
b[:7]
## 'Hello, '
```

```
b[2:]
## 'llo, World!'
```

```
b[-2]
## 'd'
```

```
b[3:6]
## 'lo,'
```

```
b[-5:-2]
## 'orl'
```

```
b[5:-2]
## ', Worl'
```

```
b[400]
## IndexError:  string index out of range
```

```
b[3.5]
## TypeError:  string indices must be integers
```

```
b[2] = 'K' # strings are immutable
## TypeError:  'str' object does not support item
assignment
```

Character string operators

```
a = "Hello"
len(a)
## 5
```

```
"char " + "string " + a      # concatenate / chain strings
## 'char string Hello'
```

```
"char " + "string" + 77
## TypeError:  can only concatenate str (not "int") to str
```

```
3 * a                                    # repeat strings
## 'HelloHelloHello'
```

```
3 / a
## TypeError:  unsupported operand type(s) for /:  'int'
and 'str'
```

```
"ell" in a
## True
```

Split and join character strings

```
char = "some text and words"
char.split() # split at spaces, return list with strings
## ['some', 'text', 'and', 'words']
char # immutable: not changed
## 'some text and words'
```

```
print("char string\nline break")
## char string
## line break
```

```
"char string\nline break".split(" ")
## ['char', 'string\nline', 'break']
```

```
"char string\nline break".split()
# the default is _any_ type of space, including \n
## ['char', 'string', 'line', 'break']
```

```
"_".join(["list", "of", "words"])
## 'list_of_words'
```

Clean and analyze character strings

```
# strip leading white space:
" char string  ".lstrip()
## 'char string  '
```

```
# strip ws on both ends:
" char string  ".strip()
## 'char string'
```

```
# strip trailing white space:
" char string  ".rstrip()
## ' char string'
```

```
# strip given symbols:
"k skates".strip("ks ")
## 'ate'
```

```
"CharString".startswith("Ch")
## True
```

```
"CharString".count("r")
## 2
```

```
# replace all instances (gsub):
"CharString".replace("Ch", "K")
## 'KarString'
```

```
"CharString".lower()
## 'charstring'
```

```
# location of first instance:
"interpreter".find("er")
## 3
```

```
max("CharString")
## 't'
```

```
"7" < "D" < "a"
## True
```

regular expressions (wildcards)

```
# re module (comes installed with python)
import re
```

```
re.sub('[xyz]', 'K', "abycd")
## 'abKcd'
```

Nice online regex tutorials:
factory-mind, regexone, javatpoint, rexegg, your recommendation

Formatted printing + F-strings (see also realpython.com)

```
name="Berry"  ;  age=32
"Hello %s, how are you?" %name
## 'Hello Berry, how are you?'
"%d %s cost $%.2f" % (6, 'bananas', 1.74)
## '6 bananas cost $1.74'
```

```
"Hi {}, you are {} years old" .format(name, age)
## 'Hi Berry, you are 32 years old'
"Hi {1}, you are {0} years old" .format(name, age)
## 'Hi 32, you are Berry years old'
"Hi {n}, you are {a} years old" .format(n=name, a=age)
## 'Hi Berry, you are 32 years old'
```

```
f"Hi {name}, you are {age} years old"
## 'Hi Berry, you are 32 years old'
x = 5 ; y = 10
f"five plus ten is {x+y} and not equal to {2*x+y}"
## 'five plus ten is 15 and not equal to 20'
```

Linter

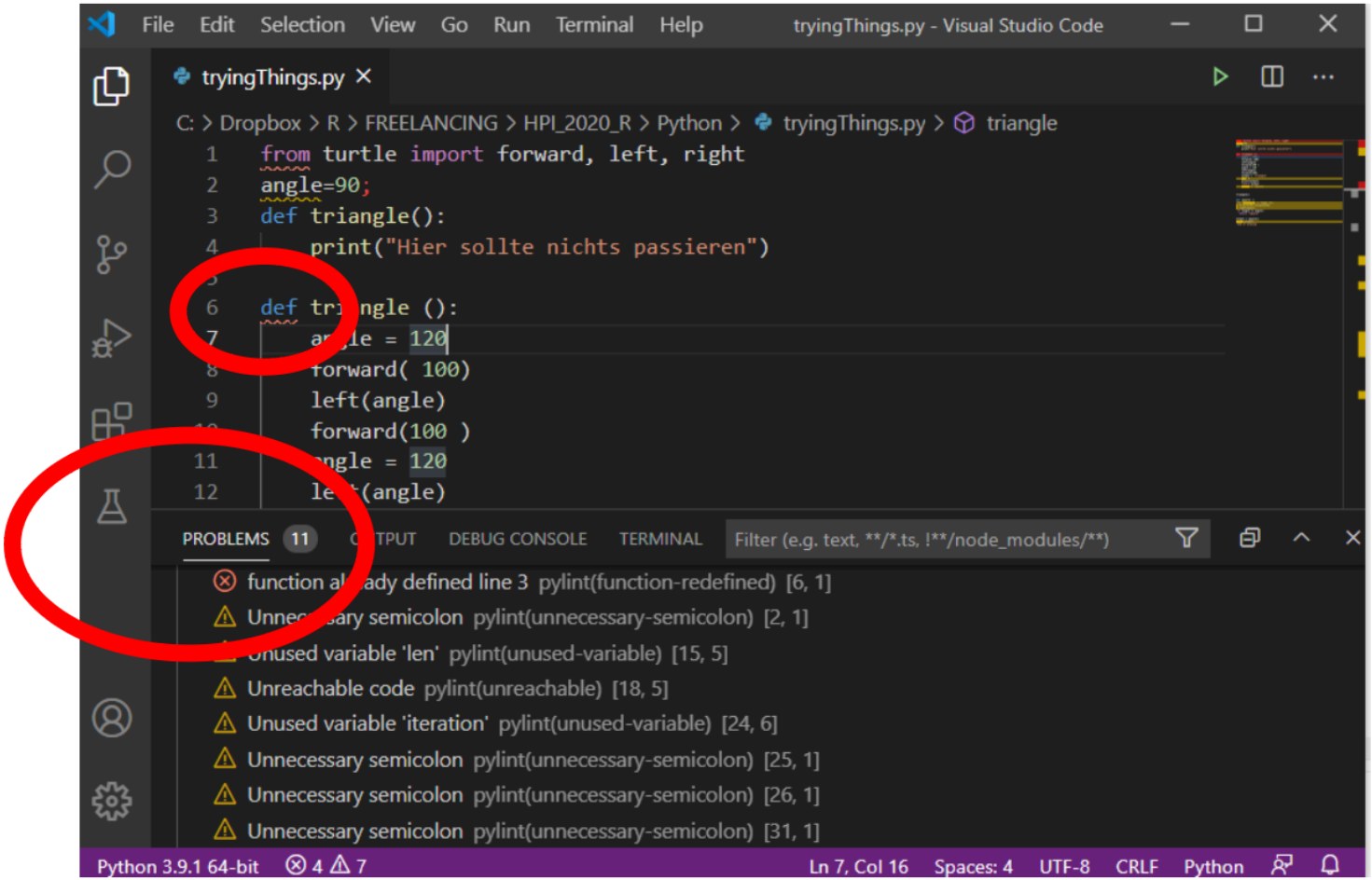
program to analyze code style and determine structural problems (pointless lines of code, potentially overwriting variable names, etc)

some messages are a bit cryptic - a few explainers:

Redefining name 'object' from outer scope (line 42)	The variable 'object' (created in line 42), is being overwritten (defined again).
EOF	End of File
EOL	End of Line
EOL while scanning string literal	A character string was opened, but not closed. A quotation mark is missing.
Unnecessary parens after 'if' keyword	if(cond): in python can simply be if cond: The brackets are not needed.
f-string expression part cannot include a backslash	`f"{Place\nholder}"` remove the \ in the placeholder code
line continuation character	Backslash \ outside of a character string
unmatched bracket	A bracket has been opened but not closed (or vice versa)
literal	A number
Unreachable code	the code will never be executed (e.g. after a return statement)
String statement has no effect	If not assigned to a variable, a charstring doesn't do anything
Catching previously caught exception type ValueError	duplicate except:-statement, remove one

Linter

use linter locally in VScode: code.visualstudio.com/docs/python/linting
for turtle, Pylint (vscode default) needs the object oriented interface:
stackoverflow.com/a/52903441
further reading: realpython.com/python-code-quality



Linter

use linter locally in RStudio:

```
install.packages("pylintR")}
```

and python package (more in 1.5 Importing):

```
pip install pylint # pip3 on Mac
```

Then: Rstudio - Addins - PYLINTR - Pylint folder (or file)

Click there, set a keyboard shortcut to it, or manually run:

```
pylintR::pylint(".")
```

For this course, I suggest to suppress some messages:

```
fname <- paste0(fs::path_home(),"/.pylintrc")
fname
cat("[MESSAGES CONTROL]
disable=invalid-name,
        missing-module-docstring,
        missing-function-docstring\n\n",
file=fname)
berryFunctions::openFile(dirname(fname))
```

see step 7 in the pylintrc doc.

Collections (Arrays)

'object types' in R In Python, not all objects are mutable:

```
alist = [1,2,3]  ;  alist
## [1, 2, 3]
```

```
alist[1] = 9  ; alist # list can be changed
## [1, 9, 3]
```

```
alist.append(77) # Method = function for an object class
alist # changed without re-assignment
## [1, 9, 3, 77]
```

type	example	mutable	ordered	indexed	duplicates
list	[1,3]	yes	yes	yes	ok
tuple	(1,2)	no	yes	yes	ok
set	{1,4}	no, but add	no	no	no
dict	{"a":7, "b":3}	yes	no/yes*	by key	no

*: since python version 3.6/3.7, dictionaries *are* ordered

tuple: group related data. **set**: quick lookup if value is contained.

Sets

```
s1 = {1,2,3,4,5}                          # operators like in
s2 = {    3,4,5,6,7,8}                    # mathematical sets
print(s1)
## {1, 2, 3, 4, 5}
```

```
s1 | s2                                                  # union
## {1, 2, 3, 4, 5, 6, 7, 8}
```

```
s1 & s2                                                 # intersection
## {3, 4, 5}
```

```
s1 - s2                              # difference: in s1, not in s2
## {1, 2}
```

```
s2 - s1
## {8, 6, 7}
```

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

Tuple

```
(4, 8, 4, -3.14)
## (4, 8, 4, -3.14)
```

List creation + subsetting I

```
a_list = [7, -4, 9, 1, 2, 3, 9, 5]
```

```
len(a_list)
## 8
```

```
a_list[0] # first element
## 7
```

```
a_list[1] # second element
## -4
```

```
a_list[5] # element 6
## 3
```

```
a_list[2:5] # elements 3,4,5
## [9, 1, 2]
```

ranges are exclusive at the right end

`:` is not an operator outside subsetting

```
empty_list = []
```

List creation + subsetting II

```
a_list
## [7, -4, 9, 1, 2, 3, 9, 5]
```

```
a_list[4:] # slicing: fifth till last element
## [2, 3, 9, 5]
```

```
a_list[:6] # 1-6th
## [7, -4, 9, 1, 2, 3]
```

```
a_list[-2] # second-to-last element
## 9
```

```
a_list[2:-3] # range from left + right
## [9, 1, 2]
```

```
a_list[2] = "newvalue" # overwrite third element
a_list
## [7, -4, 'newvalue', 1, 2, 3, 9, 5]
```

mixed data types possible

Remove list element at index

```
a_list = [1,2,3,4,4,5,6,4,4]
```

```
lastval = a_list.pop() # remove last element + return it
lastval
## 4
a_list # changed even without re-assigning
## [1, 2, 3, 4, 4, 5, 6, 4]
```

```
a_list.pop(3) # remove (+ return) selected element
a_list
## 4
## [1, 2, 3, 4, 5, 6, 4]
```

```
del(a_list[2]) # remove element at given index
a_list
## [1, 2, 4, 5, 6, 4]
```

Remove list element by value

```
a_list
## [1, 2, 4, 5, 6, 4]
```

```
4 in a_list # check for presence, return a boolean
## True
```

```
9 not in a_list # check for absence
## True
```

```
a_list.index(4) # location of first instance of 4
## 2
```

```
a_list.remove(4) # remove first instance of 4
a_list
## [1, 2, 5, 6, 4]
```

adding elements

```
a_list.append(66) # add at end
a_list
## [1, 2, 5, 6, 4, 66]
```

```
a_list.insert(3, "new_val") # insert at given position
a_list
## [1, 2, 5, 'new_val', 6, 4, 66]
```

insert moves up the elements following the insert location

```
a_list + ["L with", 3, "things"]
## [1, 2, 5, 'new_val', 6, 4, 66, 'L with', 3, 'things']
```

Copying lists

```
another_list = a_list
another_list
## [1, 2, 5, 'new_val', 6, 4, 66]
```

```
a_list.append(4)
```

```
another_list # has also changed!
## [1, 2, 5, 'new_val', 6, 4, 66, 4]
```

```
id(a_list) == id(another_list) # same obj on disc
## True
```

```
third_list = a_list.copy() # independent copy
third_list
## [1, 2, 5, 'new_val', 6, 4, 66, 4]
```

```
a_list[1] = 99
```

```
third_list # has not changed
## [1, 2, 5, 'new_val', 6, 4, 66, 4]
```

Copying nested lists: deep copy

```
DC1 = [1, 2, [3,4], 5]
DC2 = DC1
DC3 = DC1.copy()
import copy
DC4 = copy.deepcopy(DC1)
## --initially--
##  DC1: [1, 2, [3, 4], 5]
##  DC2: [1, 2, [3, 4], 5]
##  DC3: [1, 2, [3, 4], 5]
##  DC4: [1, 2, [3, 4], 5]
DC1[1] = 9
## --second element modified--
##  DC1: [1, 9, [3, 4], 5]
##  DC2: [1, 9, [3, 4], 5]
##  DC3: [1, 2, [3, 4], 5]
##  DC4: [1, 2, [3, 4], 5]
DC1[2][0] = 8
## --nested element modified:--
##  DC1: [1, 9, [8, 4], 5]
##  DC2: [1, 9, [8, 4], 5]
##  DC3: [1, 2, [8, 4], 5]
##  DC4: [1, 2, [3, 4], 5]
```

Scoping in functions (source)

```
# reduce a list before computations
def maxWithout5(ll):
    ll.remove(5)
    return max(ll)
# this will change the global list (mutable object):
list_a = [1,2,3,4,5]
print(list_a)                # [1, 2, 3, 4, 5]
print(maxWithout5(list_a)) # 4
print(list_a)                # [1, 2, 3, 4]   <-- changed!
```

```
def maxWithout5(ll):
    ll = ll.copy()           # create a local copy
    ll.remove(5)             # with a different id
    return max(ll)
list_b = [1,2,3,4,5]
print(list_b)                # [1, 2, 3, 4, 5]
print(maxWithout5(list_b)) # 4
print(list_b)                # [1, 2, 3, 4, 5]
```

List sorting

```
j_list = [7, -4, 9, 1, 2, 3]
j_list.reverse() ; j_list
## [3, 2, 1, 9, -4, 7]
```

```
j_list.sort() ; j_list
## [-4, 1, 2, 3, 7, 9]
```

```
j_list.sort(reverse=True) ; j_list
## [9, 7, 3, 2, 1, -4]
```

```
k_list = [7, -4, "9", 1, 2, 3]
k_list.sort() ; k_list
## TypeError:  '<' not supported between instances of
'str' and 'int'
```

List nesting + extending

```
m_list = [1, 2, 3, [31,32,33], 4] # Nesting possible
```

```
ch_list = ["words","with","partially","many","letters"]
ch_list[2][5] # consecutive indexing: letter 6 in word 3
## 'a'
```

```
lx = [1, 2, 3, 4]
ly = [5, 6]
lz = [7, 8, 9]
lx.extend(ly) # R: lx <- c(lx, ly)          # changes lx
lx
## [1, 2, 3, 4, 5, 6]
```

```
lx + ly # R: c(lx, ly)                  # does not change lx
## [1, 2, 3, 4, 5, 6, 5, 6]
```

```
lz * 2 # R: rep(lz, 2)
## [7, 8, 9, 7, 8, 9]
```

list printing

```
words = ["these", "are", "words"]
print(words)
## ['these', 'are', 'words']
```

```
"\n".join(words)
## 'these\nare\nwords'
```

```
print("\n".join(words))
## these
## are
## words
```

```
# splat operator unpacks list
print(*words, sep="\n")
## these
## are
## words
```

```
mixed = ["char and", 99]
```

```
print("\n".join(mixed))
## TypeError:  sequence
item 1:  expected str
instance, int found
```

```
print(*mixed, sep="\n")
## char and
## 99
```

```
isinstance( *[words,list] ) # unpack 1 list to 2 params
## True
```

reversing lists (task)

Correct the error in the code below. Why does it occur?

```
values = ["a1", "b1", "b2", "b3", "b4", "b5", "c1", "d1"]
values = values.reverse()
print(values[2]) # should be "b5"
## TypeError:  'NoneType' object is not subscriptable
```

reversing lists (solution)

The list method 'reverse' changes the list itself since it's a mutable object.
It is called for this side effect, hence "doesn't return anything"
It actually returns `None`, which overwrote 'values'.

```
values = ["a1", "b1", "b2", "b3", "b4", "b5", "c1", "d1"]
values.reverse()
print(values[2])
## b5
```

Dictionaries

- used to store data values in key:value pairs.
- mapping: associative memory, associative field (Hash in Perl)
- optimized to (very quickly) retrieve values when the key is known, even for very large datasets.
- keys must be unique, values can be whatever type

Usage examples:

- Encodings in a survey
- Patient IDs with further information (can be a list or a sub-dictionary)
- HTTP-Statuscodes
- Contact list (address book)
- Timestamp and recorded performance of machines

dict

```
trainer = {'name': "Berry", 'age' : 32}
```

```
len(trainer)
## 2
```

```
trainer['age'] = 33 ; trainer # overwrite entry
## {'name': 'Berry', 'age': 33}
```

```
trainer['meaning'] = 42 ; trainer # add entry
## {'name': 'Berry', 'age': 33, 'meaning': 42}
```

I use `'` apostrophes to access dictionaries, as they can be used in f-strings with `"` quotation marks.

```
f"Create a string with {3+4} computations."
## 'Create a string with 7 computations.'
```

```
# f-string double and single quotes cannot be mixed
f"Hi {trainer['name']}, your age is {trainer['age']}."
## 'Hi Berry, your age is 33.'
```

accessing dictionary

failsafe selection of a key:

```
trainer.get('name', "value_if_key_not_present")
## 'Berry'
```

```
trainer.get('NAME', "value_if_key_not_present")
## 'value_if_key_not_present'
```

```
trainer = {'name': "Berry", 'age' : 32}
```

```
trainer.keys()
## dict_keys(['name', 'age'])
```

```
trainer.values() # .items for looping
## dict_values(['Berry', 32])
```

```
"age" in trainer.keys()
## True
```

```
"age" in trainer # shorter and faster :)
## True
```

removing from dictionary

```
trainer = {'name':"Berry", 'age':32, 'meaning':42, 'z':0}
del(trainer['age']) # delete pair (full entry)
trainer
## {'name': 'Berry', 'meaning': 42, 'z': 0}
del trainer['z'] # brackets are optional
```

```
the_old = trainer.pop('meaning')
the_old    ;   trainer
## 42
## {'name': 'Berry'}
```

```
the_age = trainer.pop('age') # key no longer in dict
## KeyError:  'age'
trainer.pop('age', None) # invisibly returns None
```

```
trainer.clear() # remove all entries
trainer
## {}
```

printing dictionary

```
trainer = {'name': "Berry", 'age' : 32}
for k, v in trainer.items():
    print(k, v)
## name Berry
## age 32
```

dictionary: pointers and copies

```
dict1 = {'a': 1, 'b': 2, 'c': 3}
```

```
dict2 = dict1 # dict2 is only a pointer to dict1
```

```
dict3 = dict1.copy() # distinct independent object
```

If we change `dict1`, `dict2` will also be changed. (This applies to all mutable objects).

```
dict1['c'] = 333
dict2
## {'a': 1, 'b': 2, 'c': 333}
```

```
dict3
## {'a': 1, 'b': 2, 'c': 3}
```

updating dictionaries

```
dict1 = {'a': 1, 'b': 2, 'c': 3}
```

```
dict2 = {'a': 11, 'd': 4}
```

```
dict1.update(dict2) # updates values or adds key-value
pairs
```

```
print(dict1)
## {'a': 11, 'b': 2, 'c': 3, 'd': 4}
```

Logical operators

```
a == b # Equals
a != b # Not equal to
a < b  # Less than
a <= b # Less than or equal to
a > b  # Greater than
a >= b # Greater than or equal to
```

```
7 < 8 ; "9" < "A" ; "A" < "B" ; "A" < "a" ; "a" < "b"
# all True: lowercase > UPPERCASE
```

```
7 < 8 < 9 # convenient (in R: 7<8 & 8<9)
## True
```

```
not False
True and 6 > 8
9 > 8 or False
## True
## False
## True
```

Operator precedence and background conversion

In R we have:

```
7>1 & 6>1
## [1] TRUE
```

In Python we get:

```
7>1 & 6>1
## False
```

`&` is a bitwise operator.

The logical operator is `and`

```
7>1 and 6>1
## True
```

`&` has higher precedence than comparisons (is done first):

```
7 > 1&6 > 1
## False
```

```
(7>1) & (6>1)
## True
```

In logial operators, R converts both sides to logical before computing:

```
  c(-1,   0,    1,    2)  &  TRUE
## [1]  TRUE FALSE  TRUE  TRUE
```

Python does not do `bool(0) and True`:

```
-1 and True  ;  0 and True  ;  1 and True  ;  2 and True
# True              0              True          True
```

Caret

The caret (^) is a bitwise XOR (exclusive OR). It results to True (1) if one (and only one) of the operands evaluates to True.

```
0^0 # -> 0
1^1 # -> 0
1^0 # -> 1
0^1 # -> 1
8^3 # -> 11
```

because

```
1000 # 8 (binary # decimal)
0011 # 3 (binary # decimal)
---- # APPLY XOR (vertically)
1011 # result = 11 (decimal)
```

stackoverflow / What does the caret operator do

Conditional code execution: general structure

```
a = 10  ;  b = 20
if b > a:
   print("b is greater than a")
## b is greater than a
```

```
a = 30  ;  b = 30
if b > a:
   print("b is greater than a")
elif a == b:
   print("a and b are equal")
## a and b are equal
```

```
a = 100  ;  b = 30
if b > a:
   print("b is greater than a")
elif a == b:
   print("a and b are equal")
else:
   print("a is greater than b")
## a is greater than b
```

Conditional code execution: combining conditions

```
a = 100   ;   b = 30
if a>b and (b==20 or b==30):
   print("a is bigger  and  b is 20 or 30")
## a is bigger  and  b is 20 or 30
```

```
cond = [True, False, True, True, True]
all(cond)
any(cond)
sum(cond)
## False
## True
## 4
```

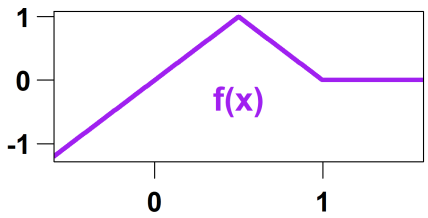
Conditional code - math example

mathematical function with case differentiation

$$f_n(x) = \begin{cases} 2n^2x & , 0 \le x \le \frac{1}{2n} \\ n-2n^2(x-\frac{1}{2n}) & , \frac{1}{2n} \le x \le \frac{1}{n} \\ 0 & , \frac{1}{n} \le x \le 1 \end{cases}$$

source with n=1 & `inf` outer boundaries:

```
        2x            for x < 0.5
f(x) = 1-2(x-0.5)     for 0.5 <= x <= 1
        0             for x > 1
```




```
x = -0.2 # -0.4 ; x = 0.6 # 0.8 ; x = 1.6 # 0
if x>1:
  y = 0
elif x>= 0.5:
  y = 2-2*x
else:
  y = 2*x
print(y)
## -0.4
```

Conditional code - input example

```
username = input('Enter username: ')
password = input('Enter password: ')
```

print `login successful` for user admin with pw 123456, otherwise `username or password is wrong`

```
if username == 'admin' and password == '123456':
    print('login successful')
else:
    print('username or password is wrong')
```

Conditional code - leap year example

A leap year is a calendar year that contains an additional day (Feb 29) to stay synchronized with the astronomical year. These extra days occur in each year which is an integer multiple of 4, except for years evenly divisible by 100, which are not leap years unless evenly divisible by 400.

Find out if a year is leap year.
in Python, `or` has precedence over `and`.
modulo operator is `%`.
year should be divisble by 4 and (not divisible by 100 or divisible by 400)

```
year = int(input('Enter year: '))
is_leap = year%4==0 and year%100!=0 or year%400==0
print(is_leap)
```

Loops structure

```
# General structure:
for variable in list_of_values :
    print(variable)
# colon (:) needed
# indentation matters
```

```
while cond:
    run_things()
    if cond1:
        continue # jump to the next iteration (next in R)
    if cond2:
        break # stop the loop
    888 # not run if cond2
```

convention for unused index variable:

```
for _ in range(3): # or _var
    print("stuff") # -> stuff stuff stuff
```

for loop print example 1

```
for number in (0,1,2,3):
    print(number)
## 0
## 1
## 2
## 3
```

```
for number in range(4):
    print(number)
## 0
## 1
## 2
## 3
```

```
list( range(8, 0, -2) ) # range stop exclusive
## [8, 6, 4, 2]
```

for loop print example 2

print all object names in a module, each on a separate line of output

```
import math # built-in module (package)
for f in dir(math):
    print(f)
## __doc__
## ...    # manually selected output
## __name__
## __package__
## acos
## ceil
## exp
## factorial
## gamma
## inf
## isnan
## log10
## pi
## pow
## sin
## sqrt
## trunc
```

for loop selection example

select all even numbers from a list

```
numbers = [468,976,701,269,841,7,917,698,689,526,307,791,718]
```

```
even = [] # initiate empty list
```

```
for n in numbers:
    if n%2==0:
        even.append(n)
```

```
even
## [468, 976, 698, 526, 718]
```

while loop print example

as long as x is positive, print it and decrement it by 10

```
x = 50
while x > 0:
    print (x)
    x -= 10
print("final value:", x)
## 50
## 40
## 30
## 20
## 10
## final value: 0
```

while loop input example

repeatedly `input` a number, until it is guessed correctly

```
num = 0
while num != 42 :
    num = input("guess the number: ")
    num = int(num)
    if num==42:
        print("You found the answer to life (etc)...")
    elif num > 42:                          # flush for Rstudio
        print(f"Sorry, {num} is too big.", flush=True)
    else:
        print(f"Sorry, {num} is too small.", flush=True)
## guess the number: 78
## Sorry, 78 is too big.
## guess the number: 31
## Sorry, 31 is too small.
## guess the number: 42
## You found the answer to life (etc).
```

while loop input alternative

```
while(True):
    num = input("guess the number: ")
    num = int(num)
    if num==42:
        print("You found the answer to the universe...")
        break
    elif num > 42:
        print(f"Sorry, {num} is too big.")
    else:
        print(f"Sorry, {num} is too small.")
```

`while(True)` starts a loop that must be ended manually (R: `repeat`)

Loop control examples

```
for letter in 'Python':
    if letter == 'h':
        continue                # skip the rest of an iteration
    print("Current: " + letter)
## Current: P
## Current: y
## Current: t
## Current: o
## Current: n
```

```
for letter in 'Python':
    if letter == 'h':
        break                   # terminate the loop completely
    print("Current: " + letter)
## Current: P
## Current: y
## Current: t
```

mapping

```
charlist = ["abcdef", "ab", "abc"]
charlen = []
for c in charlist:
  charlen.append(len(c))
charlen
## [6, 2, 3]
```

```
myclen = map(len, charlist) # lapply in R
myclen
## <map object at 0x000001CCC9178D90>
list(myclen)
## [6, 2, 3]
```

```
list_a = [1, 2, 3]
list_b = [10, 20, 30]
list(  map(lambda x, y: x + y, list_a, list_b)  )
## [11, 22, 33]
```

anonymous function

handle several objects in a single line of code, see 1.5 Functions.

```
for a,b in ((1,11), (2,22), (3,33), (4,44)):
    print("a:", a, " b:", b, " result: ", b-2*a)
## a: 1  b: 11  result:  9
## a: 2  b: 22  result:  18
## a: 3  b: 33  result:  27
## a: 4  b: 44  result:  36
```

Changing objects in loops 1

Don't change the iterator object during a loop
e.g. don't conditionally remove elements

```
ids = ["a1","b1","b2","b3","b4","b5","b6","c1","d1"]
# remove values starting with b
for v in ids:
  if v[0]=="b": ids.remove(v)
ids
## ['a1', 'b2', 'b4', 'b6', 'c1', 'd1']
```

Why are half the b's still left?

After the second iteration, ids is `a1, b2, b3, b4, b5, b6, c1, d1`, because "b1" has been removed. In the third iteration, Python evaluates 'ids' again and passes the third element to `v` ("b3"). This gets removed again, so we have `a1, b2, b4, b5, b6, c1, d1`. Then this repeats until we are left with `a1, b2, b4, b6, c1, d1`. If we were looping over an index with `for i in range(len(ids))`, we would get an IndexError.

How can we correctly select values starting with b in a loop?

Changing objects in loops 2

```
ids = ["a1","b1","b2","b3","b4","b5","b6","c1","d1"]
output = []
for v in ids:
  if v[0]!="b": output.append(v)
print(output)
## ['a1', 'c1', 'd1']
```

Unexpected things: iteration in python

There is an alternative without any (apparent) loop here as well:

```
list(filter(lambda x : x[0]!="b", ids))
## ['a1', 'c1', 'd1']
```

Changing objects in loops 3

Note that for loops work differently in R. There, 'ids' is evaluated once and then the loop is run. It is not re-evaluated for every iteration.

```r
# Mimicking .remove from Python:
ids <- c("a1", "b1", "b2", "b3", "b4", "b5", "b6", "c1", "d1")
for(v in ids)
  if(substr(v,1,1)=="b") ids <- ids[-which(ids==v)[1]]
ids # "a1" "c1" "d1"

# Real life R usage (vectorized):
ids <- c("a1", "b1", "b2", "b3", "b4", "b5", "b6", "c1", "d1")
ids[substr(ids,1,1)!="b"] # "a1" "c1" "d1"
```

iterators

```
for l in "hello":
    print(l)
## h
## e
## l
## l
## o
```

Strings are iterables = objects that can be iterated (looped) over

```
iter("hello")
## <str_iterator object at 0x000001CCC917B580>
```

Iterator with a `__next__` method

Sub-kind of iterator, adding a counter to an iterable:

```
enumerate("hello")
## <enumerate object at 0x000001CCC915AC80>
```

stackoverflow post

for loop max example

```
values = [922,790,447,617,534,93,895,60,21,
          962,992,302,435,902,795,482]
```

Imagine the function `max` is not available.

Using a loop, determine the largest value.

```
maxi = 0
for v in values:
    if v > maxi:
        maxi = v
print(maxi)
## 992
```

List comprehension example 1

reduce code length

```
def do_something_with(x):
    return x + 5
item_list = [6, 9, 17, -2, 24]
```

```
result = []
for item in item_list:
  new_item = do_something_with(item)
  result.append(new_item)
result
## [11, 14, 22, 3, 29]
```

```
result = [do_something_with(item) for item in item_list]
result
## [11, 14, 22, 3, 29]
```

```
list(map(do_something_with, item_list))
## [11, 14, 22, 3, 29]
```

List comprehension example 2

```r
# in R: select vector entries starting with
char_vec <- c("Alex","Berry","Bethany","Chris","Dave")
char_vec[substr(char_vec,1,1)=="B"]
## [1] "Berry"   "Bethany"
grep("^B", char_vec, value=TRUE)
```

```python
charstring_list = ["Alex","Berry","Beth","Chris","Dave"]
```

```python
withB = []
for word in charstring_list:
  if word[0] == "B":
     withB.append(word)
withB
## ['Berry', 'Beth']
```

```python
withB = [x for x in charstring_list if x[0] == "B"]
withB
## ['Berry', 'Beth']
```

dictionary comprehension works the same way

```
# Get number of letters in each word:
message = "your programming improves"
{w:len(w) for w in message.split()}
## {'your': 4, 'programming': 11, 'improves': 8}
```

```
# Double each value in the dictionary:
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f':6}
double_dict1 = {k:v*2 for (k,v) in dict1.items()}
```

```
# Select the entries >2 and multiples of 2:
{k:v for (k,v) in dict1.items() if v>2 if v%2==0}
## {'d': 4, 'f': 6}
```

```
# Convert degrees Fahrenheit to Celcius:
dF = {'t1': -30,'t2': -20,'t3': -10,'t4': 0}
dC = {k:(float(5)/9)*(v-32) for (k,v) in dF.items()}
dC = {k+"C":round(v) for (k,v) in dC.items()}
dC
## {'t1C': -34, 't2C': -29, 't3C': -23, 't4C': -18}
```

tuple vs generator

```
numbers = [1, 2, 3, 5, 7]
squares = [n**2 for n in numbers] # list object
squares
## [1, 4, 9, 25, 49]
sum(squares)
## 88
sum(tuple(squares)) # works
## 88
```

```
numbers = [1, 2, 3, 5, 7]
squares = (n**2 for n in numbers)
squares # generator, not tuple
## <generator object <genexpr> at 0x000001CCC4172EA0>
sum(squares)
## 88
sum(tuple(squares)) # empty tuple
## 0
```

List comprehension exercise

Abbreviate each of the following 3 loops to a single line of code

```
values = [11,10,2,3,15,3,5,7,7,2,8,7,5,6,5,8,5,9,6,3,15,6,9]
sum_square_values = 0
for v in values:
   sum_square_values += v**2
sum_square_values
## 1351
```

```
numbers = [951,402,984,651,360,69,408,319,601,485,980,507,725,
547,544,615,83,165,141,501,263,617,865,575,219,390,984,592,236,
105,942,941,386,462,47,418,907,344]
even_numbers = []
for n in numbers:
   if n%2==0: even_numbers.append(n)
even_numbers
## [402, 984, 360, 408, 980, 544, 390, 984, 592, 236, 942, 386, 462, 418, 344]
```

```
import random
max_exp = []
for i in range(50):
   max_exp.append(random.expovariate(0.2))
max_exp = max(max_exp)
max_exp
## 28.648948267657826
```

List comprehension exercise solutions

```
values = [11,10,2,3,15,3,5,7,7,2,8,7,5,6,5,8,5,9,6,3,15,6,9]
sum_square_values = sum([v**2 for v in values]) ; sum_square_values
## 1351
```

```
numbers = [951,402,984,651,360,69,408,319,601,485,980,507,725,
547,544,615,83,165,141,501,263,617,865,575,219,390,984,592,236,
105,942,941,386,462,47,418,907,344]
even_numbers = [n for n in numbers if n%2==0] ; even_numbers
## [402, 984, 360, 408, 980, 544, 390, 984, 592, 236, 942, 386, 462, 418, 344]
```

```
import random # built-in python module (no installation needed)
max_exp = max([random.expovariate(0.2) for _ in range(50)]) ; max_exp
## 15.434094958612917
```

Error management: try - except

TypeError: wrong data type for operator or function

```
7 + "2"
## TypeError:  unsupported operand type(s) for +:  'int'
and 'str'
```

upon suspected error, don't fail the entire script execution, just print a message.
Note the indentation as with all python control structures.

```
try:
    7 + "2" # code that might fail
except TypeError:
    print("That mixed charstrings and numbers")
## That mixed charstrings and numbers
```

generic except

```
try:
    7 + nonexistingobject
except TypeError:
    print("That mixed charstrings and numbers")
except:
    print("Another error occured")
## Another error occured
```

for any type of error, don't misuse

```
try:
    7 + "2" # code that may fail
except:
    print("Code failed")
else:
    print("Code succeeded")
## Code failed
```

The `else` code is run if there was no error. Could be in the `try` part, but keep potential failure + handling close together, handle only that error (keep unexpected errors).
Also, the `else` code is run before `finally` (next slide).

finally

```
try:
    7 + "2" # code that may fail
except:
    print("Code failed")
finally:
    print("Program finished")
## Code failed
## Program finished
```

Code in `finally` is run even if return / break / continue is called or another (uncaught) exception is raised

traceback will tell you where the error comes from:

```
import traceback
try:
   7 + nonexistingobject
except:
   print("Error occured:",traceback.format_exc())
## Error occured: Traceback (most recent call last):
##   File "<string>", line 2, in <module>
## NameError: name 'nonexistingobject' is not defined
```

More informative in real life (slide structure doesn't support traceback).

`script1.py` line 5 calls `sc2.py` line 234 calls module `statistics`

IDEs with a debugger will often provide tracebacks for errors.

exception object

log errors with custom prefix

```
def fail_with_message(x):
  try:
    x + 7
  except Exception as e: # bind exception as variable 'e'
    print("An error occured:",e,sep="\n")#\n: slide width
```

`Exception`: All exceptions are derived from this class

```
fail_with_message(3)
fail_with_message(None)
## An error occured:
## unsupported operand type(s) for +: 'NoneType' and 'int'
fail_with_message("3")
## An error occured:
## can only concatenate str (not "int") to str
```

Exercise: `fail_with_message(3)` should yield 10

exception info

```
def fail_with_full_message(x): # see also sys.exc_info()
  try:
    x + 7
  except Exception as e:
    print(f"A {type(e).__name__} occured:\n{e}")
```

```
fail_with_full_message(3)
fail_with_full_message(None)
## A TypeError occured:
## unsupported operand type(s) for +: 'NoneType' and 'int'
fail_with_full_message("3")
## A TypeError occured:
## can only concatenate str (not "int") to str
fail_with_full_message(dummyvar)
## NameError:  name 'dummyvar' is not defined
```

messages with time stamp, useful for logging

```
import time
now = time.strftime("%Y-%m-%d %H:%M UTC", time.gmtime())
print(now)
## 2023-01-25 18:24 UTC
```

```
def fail_with_timestamp(x):
  try:
    x + 7
  except:
    n = time.strftime("%Y-%m-%d %H:%M UTC",time.gmtime())
    print("An error occured at", n)
```

```
fail_with_timestamp(3)
fail_with_timestamp(None)
## An error occured at 2023-01-25 18:24 UTC
```

Error management example

```
price = None
while not price:
  try:
    price = int(input("Enter a price: "))
  except ValueError:
    print("Please enter a valid number.")
print("The entered number was: ", price)
```

A few term definitions:
object: collection of data (variables) and methods (functions) that act on those data. (attributes + behaviour)
class: blueprint for the object, it contains a set of characteristics
instance: specific object with its own set of data and behaviors

```
class Person:
    pass
```

Python does not permit an empty body in classes and loops, so the `pass` statement is added. It acts as a placeholder for future code.

A custom class (UpperCamelCase) enables custom methods for objects.

```
p1 = Person() # create object instance
p1.name = "Berry" ; p1.age = 32 # add attributes individually
p1
## <__main__.Person object at 0x000001A7E6383F70>
p1.__dict__ # dictionary of all given params and args
## {'name': 'Berry', 'age': 32}
```

define a custom class

```
class Person:
    def __init__(self, name, age): # __init__ method
        self.name = name
        self.age = age
```

```
person1 = Person('Harry', 25)    # instantiation
person2 = Person('Hillary', 16)
person1.name
## 'Harry'
```

__**init**__: special function to initialize (assign values) to the data members of the class when an object of class is created. It is run as soon as an object of a class is instantiated.

self: represents an object of a given class that we create. first argument of non-static methods like the function `__init__`.

Objects `person1` and `person2` have their own attributes (name, age). With the self argument, the class Person is able to hold these information for all objects.

methods in a custom class

Add a method to assess whether a person is allowed to watch horror movies (if older than 18)

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def can_watch_horror_movie(self):    # this function
        if self.age >= 18:               # will be a method
            return "go ahead"            # for all objects
        else:                            # of class 'Person'
            return "sorry, too young"
```

```
person1 = Person('Harry', 25)
person1.can_watch_horror_movie()
## 'go ahead'
person2 = Person('Hillary', 16)
person2.can_watch_horror_movie()
## 'sorry, too young'
```

In the `Person` class, how can we simplify the `can_watch_horror_movie` method to return True / False?

```
def can_watch_horror_movie(self):
    return self.age >= 18
```

class with input checks

```
class Patient:
    def __init__(self, pid, gender, BP):
        self.pid = pid
        self.gender = gender
        self.BP = BP
        if gender not in ["Male","Female","Other"]:
            raise ValueError("Gender must be in M/F/Other")
    def is_hypertensive(self):
        return self.BP > 130
```

```
Patient(pid="Pat456", gender="Diverse", BP=120)
## ValueError:  Gender must be in M/F/Other
```

```
poordude = Patient("Pat123", "Male", 113)
poordude.BP = 150
print("needs attention:", poordude.is_hypertensive())
## needs attention: True
```

Further reading

Dr Philip Yip, Python101, and more

Class variables

```
class Student:
    school = "HPI" # Class variable
    def __init__(self, name):
        self.name = name
    def info(self):
        s = f"which has {len(self.school)} letters."
        print(f"{self.name} is at {self.school}, {s}")
```

Dr Philip Yip, pynative

```
s1 = Student("Berry")
s1.school # like a default argument for function parameter
## 'HPI'
s1.info()
## Berry is at HPI, which has 3 letters.
```

```
s1.school = "New school"
s1.info()
## Berry is at New school, which has 10 letters.
```

```
s2 = Student("Christina")
s2.school
## 'HPI'
```

protected / private objects

```
class Student:
    _greeting = 'Hello' # protected (by convention)
    __schoolName = 'HPI' # private class attribute
    def __init__(self, name):
        self.__name=name  # private instance attribute
    def display(self):
     print(f'{self.__name} is at {self.__schoolName}.')
```

```
std = Student("James")
std._greeting # can be accessed, but shouldn't
std.__schoolName # AttributeError: 'Student' object
std.__name       # has no attribute '__schoolName'
```

```
std.display() # James is at HPI.
```

```
std.__name = "Elise" # No error
std.display() # but this still prints James
```

class exercise (task + partial solution)

Heron's formula gives the area of a triangle when the length of all three sides is known. Any side of a triangle cannot be longer than the sum of the other two.

semiperimeter `s` = (a+b+c)/2 area `a` = sqrt(s(s-a)(s-b)(s-c))

Write code giving the area - or "Not a triangle" if a side is too long

```
a = 10
b = 4
c = 5
if a+b > c  and  a+c > b  and  b+c >a:
    s = (a+b+c)/2
    area = (s*(s-a)*(s-b)*(s-c))**0.5  #**0.5 = sqrt
    f'Area of triangle is {area}'
else:
    'Not a triangle'
```

Write a `Triangle` class that raises an error if an invalid triangle is created. It should have an area method built in.

class exercise (solution)

```
class Triangle:
  def __init__(self, a, b, c):
    self.a = a
    self.b = b
    self.c = c
    if not(a+b > c and a+c > b and b+c >a):
        raise ValueError('invalid triangle. long side')
  def area(self):
    p = (self.a+self.b+self.c)/2
    area = (p*(p-self.a)*(p-self.b)*(p-self.c))**0.5
    return area

tri1 = Triangle(a=3, b=4, c=5)
tri1.__dict__ # print attributes, tri1.__dict__['c']
tri1.area()
## {'a': 3, 'b': 4, 'c': 5}
## 6.0
```

Numpy

computationally efficient numerical arrays

vectorized operations + datatypes (matrix/array) as in R

```
pip install numpy # shell, Mac pip3
```

```
import numpy as np
```

```
np.array([42, 77, 3, -26]) # list to array
## array([ 42,  77,   3, -26])
```

```
ar1 = np.arange(1, 10) # sequential 1D array
ar1
## array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
type(ar1)
## <class 'numpy.ndarray'>
```

```
ar1.size # length
## 9
```

```
ar1.sum()        # alternative:  np.sum(ar1)
## 45
```

Attributes are accessed without brackets, methods with brackets.

1D array subsetting

```
ar1
## array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
ar1[4] ; ar1[-1] # fifth and last element of 1D array
## 5
## 9
```

```
ar1[:5] ; ar1[4:] ; ar1[4:7] # slicing as with lists
## array([1, 2, 3, 4, 5])
## array([5, 6, 7, 8, 9])
## array([5, 6, 7])
```

```
ar1[::2] ; ar1[1::3] # array[start:stop:step]
## array([1, 3, 5, 7, 9])
## array([2, 5, 8])
```

How could you use this to reverse all elements?

```
ar1[::-1] # Works for lists & charstrings as well
## array([9, 8, 7, 6, 5, 4, 3, 2, 1])
```

numpy 2D arrays

```
ar2 = np.array([[1,2,3,4,5], [6,7,8,9,0]])
```

```
ar2
## array([[1, 2, 3, 4, 5],
##        [6, 7, 8, 9, 0]])
```

```
ar2.shape
## (2, 5)
```

```
ar2.size
## 10
```

```
ar2.ndim
## 2
```

More dimensions are also possible

```
ar2.dtype # numpy-internal data type
## dtype('int32')
```

```
ar2.itemsize # number of bytes per item
## 4
ar2.nbytes # total space on disc
## 40
```

2D array subsetting

```
# random integers 0-9, 3 rows, 4 columns:
arr = np.random.randint(10, size=(3,4))
arr
## array([[3, 6, 5, 4],
##        [8, 9, 1, 7],
##        [9, 6, 8, 0]])
```

```
arr[:2, :3] # 2 rows, 3 columns
## array([[3, 6, 5],
##        [8, 9, 1]])
```

```
arr[:, 3] # all values in 4th column
## array([4, 7, 0])
```

```
arr[0] # first row, not first column. BAD syntax!
## array([3, 6, 5, 4])
```

```
arr[0, ] # clean syntax
## array([3, 6, 5, 4])
```

array changes and copies

```
arr[2,0] = 3.1415 # change single element
# If array is integer, float is silently truncated to 3:
Downcasting
```

```
arr_sub = arr[:2, :2]
arr_sub[0,0] = 99 # changes both arr_sub and arr
```

```
arr_sub = arr[:2, :2].copy()
arr_sub[0,0] = 42 # does not change arr
```

```
ar12 = np.arange(1,13)                # 1D array to 2D array
ar12.reshape((3,4))                   # does not change ar12
## array([[ 1,  2,  3,  4],
##        [ 5,  6,  7,  8],
##        [ 9, 10, 11, 12]])
```

```
ar12[np.newaxis, :]     # 2D, 1 row. does not change ar12
## array([[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]])
```

Ufunc (Universal functions operating on full array)

vectorized operations

```
ara = np.arange(11, 15)
arb = np.arange(23, 27)
```

```
1 / ara # element-wise operation: broadcasting
## array([0.09090909, 0.08333333, 0.07692308, 0.07142857])
```

```
ara ; arb
## array([11, 12, 13, 14])
## array([23, 24, 25, 26])
```

```
ara * arb
## array([253, 288, 325, 364])
```

```
ara >= 13
## array([False, False,  True,  True])
```

```
ara + np.arange(23, 29) # no vector recycling
## ValueError:  operands could not be broadcast together
with shapes (4,) (6,)
```

```
arr < 6
## array([[False, False,  True,  True],
##        [False, False,  True, False],
##        [ True, False, False,  True]])
```

```
np.any(arr<6)
## True
```

```
np.any(arr<6, axis=0)    # columns: apply(a, 2, any) in R
## array([ True, False,  True,  True])
```

```
np.any(arr<6, axis=1)              # rows: apply(a, 1, any)
## array([ True,  True,  True])
```

```
np.sum(arr<6, axis=1)    # rowSums(a<6),
apply(a<6,1,sum)
## array([2, 1, 2])
```

Pandas: panel data analysis, builds on numpy

Like R dataframes. pandas API documentation

```
pip install pandas # shell, pip3 on Mac
```

```
import numpy as np
import pandas as pd
```

```
s = pd.Series(data=[.007,42], index=["bond","univ"])
s
## bond     0.007
## univ    42.000
## dtype: float64
s.size # number of elements
## 2
```

```
s.to_list()
## [0.007, 42.0]
s.to_dict()
## {'bond': 0.007, 'univ': 42.0}
```

Pandas basics

```
s1 = pd.Series([1,2,3,4], index=["A","C","D","E"])
s2 = pd.Series([1,2,5,4], index=["A","B","C","E"])
```

```
s1
## A    1
## C    2
## D    3
## E    4
## dtype: int64
```

```
s2
## A    1
## B    2
## C    5
## E    4
## dtype: int64
```

```
s1["A"] # subset by name
## 1
```

```
s1 + s2 # Operations per index
## A    2.0
## B    NaN
## C    7.0
## D    NaN
## E    8.0
## dtype: float64
```

Pandas DataFrames

```
df = pd.DataFrame(np.random.randn(5,4),   # rnorm in R
                  index='A B C D E'.split(), # rownames
                  columns='W X Y Z'.split()) # colnames
```

```
df
##           W         X         Y         Z
## A -0.514625 -0.449554  1.734621  0.643380
## B  0.026139  0.080381 -0.797389 -0.628086
## C -0.346227  0.968085  0.705617 -2.156697
## D  0.950599  0.538189 -0.450793  0.550832
## E -0.759563  1.223760 -0.159821 -1.455847
```

```
df.shape
## (5, 4)
```

```
df.index
## Index(['A', 'B', 'C', 'D', 'E'], dtype='object')
```

```
df.columns
## Index(['W', 'X', 'Y', 'Z'], dtype='object')
```

DataFrame values + dtypes

```
df.values # as array
## array([[-0.51462472, -0.44955445,  1.73462147,  0.64338016],
##        [ 0.0261394 ,  0.08038096, -0.79738899, -0.62808643],
##        [-0.34622743,  0.96808488,  0.70561737, -2.15669677],
##        [ 0.95059946,  0.53818853, -0.45079319,  0.55083241],
##        [-0.75956338,  1.22376001, -0.15982058, -1.45584663]])
```

```
df.index.values
## array(['A', 'B', 'C', 'D', 'E'], dtype=object)
```

```
df.columns.values.tolist() # colnames(df) in R
## ['W', 'X', 'Y', 'Z']
```

```
df.dtypes
## W    float64
## X    float64
## Y    float64
## Z    float64
## dtype: object
```

DataFrame info

```
df.info() # str(df) in R
## <class 'pandas.core.frame.DataFrame'>
## Index: 5 entries, A to E
## Data columns (total 4 columns):
##  #   Column  Non-Null Count  Dtype
## ---  ------  --------------  -----
##  0   W       5 non-null      float64
##  1   X       5 non-null      float64
##  2   Y       5 non-null      float64
##  3   Z       5 non-null      float64
## dtypes: float64(4)
## memory usage: 200.0+ bytes
```

Column selection by name

```
df["W"]
## A   -0.514625
## B    0.026139
## C   -0.346227
## D    0.950599
## E   -0.759563
## Name: W, dtype: float64
```

```
df.W
## A   -0.514625
## B    0.026139
## C   -0.346227
## D    0.950599
## E   -0.759563
## Name: W, dtype: float64
```

```
df[["X","W"]]
##           X         W
## A -0.449554 -0.514625
## B  0.080381  0.026139
## C  0.968085 -0.346227
## D  0.538189  0.950599
## E  1.223760 -0.759563
```

Column selection by location

```
df.iloc[:, -1] # last column by index location
## A    0.643380
## B   -0.628086
## C   -2.156697
## D    0.550832
## E   -1.455847
## Name: Z, dtype: float64
```

```
df.iloc[:, 1:3] # Columns 2+3
##           X         Y
## A -0.449554  1.734621
## B  0.080381 -0.797389
## C  0.968085  0.705617
## D  0.538189 -0.450793
## E  1.223760 -0.159821
```

Row selection

```
df.loc["B"] # row with name B, rotated view (!)
## W    0.026139
## X    0.080381
## Y   -0.797389
## Z   -0.628086
## Name: B, dtype: float64
```

```
df.iloc[2] # 3rd row
## W   -0.346227
## X    0.968085
## Y    0.705617
## Z   -2.156697
## Name: C, dtype: float64
```

```
df.iloc[0:3] # first 3 rows
##           W         X         Y         Z
## A -0.514625 -0.449554  1.734621  0.643380
## B  0.026139  0.080381 -0.797389 -0.628086
## C -0.346227  0.968085  0.705617 -2.156697
```

Row selection, filtering

```
df.iloc[-1] # last row
## W   -0.759563
## X    1.223760
## Y   -0.159821
## Z   -1.455847
## Name: E, dtype: float64
```

```
df[ df.Y > 0 ]
##           W         X         Y         Z
## A -0.514625 -0.449554  1.734621  0.643380
## C -0.346227  0.968085  0.705617 -2.156697
```

Element selection

```
df.iloc[2, 3]
## -2.156696774281812
```

```
df.loc["B", "X"]
## 0.08038096427837299
```

```
df.Y[2:4]
## C    0.705617
## D   -0.450793
## Name: Y, dtype: float64
```

Pandas methods I

```
df
##           W         X         Y         Z
## A -0.514625 -0.449554  1.734621  0.643380
## B  0.026139  0.080381 -0.797389 -0.628086
## C -0.346227  0.968085  0.705617 -2.156697
## D  0.950599  0.538189 -0.450793  0.550832
## E -0.759563  1.223760 -0.159821 -1.455847
```

```
df.Y.shape # attribute
## (5,)
```

```
df.Y.sum() # method
## 1.0322360723584882
```

```
sum(df.Y) # also possible
## 1.0322360723584882
```

```
df.sum()
## W   -0.643677
## X    2.360860
## Y    1.032236
## Z   -3.046417
## dtype: float64
```

Pandas methods II

```
df2 = np.random.default_rng(123) # with seed
df2 = df2.exponential(1e4, size=[4,6])
df2 = pd.DataFrame(df2).astype(int).astype(str)
df2
##         0     1     2     3      4      5
## 0    5969  1170  2517  3079    918  23516
## 1   42884  1546  4977  8262   8893   3647
## 2   16303  4682  7597  8091  11912   1462
## 3    5039  2769  3781  5794  15323  11195
```

```
df2[4].str.len() # only for series (column), not for df
## 0    3
## 1    4
## 2    5
## 3    5
## Name: 4, dtype: int64
```

Pandas DataFrame from dictionary

```
mydict = {'physician':["Smith", "Brown", "Williams"],
          'rate':[76, 153, 94]}
```

```
pd.DataFrame(mydict)
##   physician  rate
## 0     Smith    76
## 1     Brown   153
## 2  Williams    94
```

Missing values in pandas

```
import numpy as np
import pandas as pd
```

```
df1 = pd.DataFrame({'A':[1, None, np.nan],
                    'B':[5, 2, np.nan], 'C':[1, 2, 3]})
```

```
df1
##      A    B  C
## 0  1.0  5.0  1
## 1  NaN  2.0  2
## 2  NaN  NaN  3
```

```
df1.isna()
##         A      B      C
## 0  False  False  False
## 1   True  False  False
## 2   True   True  False
```

```
# Number of Nas per column
df1.isna().sum()
## A    2
## B    1
## C    0
## dtype: int64
```

```
# number of Nas per row
df1.isna().sum(axis=1)
## 0    0
## 1    1
## 2    2
## dtype: int64
```

Dropping NAs

```
df1[df1.B.isna()].index.tolist()
## [2]
```

```
df1[df1["B"].notna()]
##      A    B  C
## 0  1.0  5.0  1
## 1  NaN  2.0  2
```

```
df1.dropna()
##      A    B  C
## 0  1.0  5.0  1
```

```
df1.dropna(axis=1)
##    C
## 0  1
## 1  2
## 2  3
```

```
# >=2 finite numbers
# needed to be kept
df1.dropna(thresh=2)
##      A    B  C
## 0  1.0  5.0  1
## 1  NaN  2.0  2
```

Imputing NAs

```
df1
##      A    B  C
## 0  1.0  5.0  1
## 1  NaN  2.0  2
## 2  NaN  NaN  3
```

```
df1.fillna(value='missing') # replace NA with "missing"
##           A        B  C
## 0       1.0      5.0  1
## 1   missing      2.0  2
## 2   missing  missing  3
```

changes dtype of the column!

```
# Replace with mean value of column:
df1.B.fillna(value=df1.B.mean())
## 0    5.0
## 1    2.0
## 2    3.5
## Name: B, dtype: float64
```

Reading dataframes from files

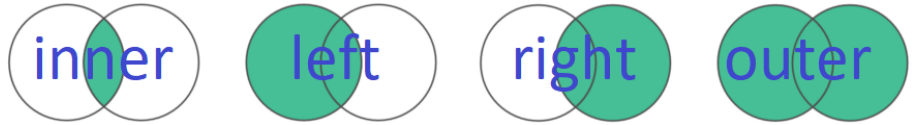
```
import pandas as pd
```

```
df = pd.read_csv("file.txt") # read_table with sep="\t"
df = pd.read_excel("file.xls") # also "file.ods"
```

pandas read functions overview + details

```
import io    # multiline string for line breaks
data = """
a,b,c,d
1,2,3,4
5,6,7,8
9,0,1  """
df = pd.read_csv(io.StringIO(data)) # fills with NAs
print(df)
##    a  b  c    d
## 0  1  2  3  4.0
## 1  5  6  7  8.0
## 2  9  0  1  NaN
```

Merging dataframes




```
import pandas as pd
df1 = pd.DataFrame({'color': ['red','red','tan','tan'],
                    'size' : [  245,  260,  189,  205]})
df2 = pd.DataFrame({'color': ['blue','red','grey'],
                    'shape': [   'A',  'A',   'B']})
```

```
pd.merge(df1, df2) # inner
##   color  size shape
## 0   red   245     A
## 1   red   260     A
```

```
pd.merge(df1,df2, how="outer")
##   color   size shape
## 0   red  245.0     A
## 1   red  260.0     A
## 2   tan  189.0   NaN
## 3   tan  205.0   NaN
## 4  blue    NaN     A
## 5  grey    NaN     B
```

```
pd.merge(df1, df2, "left")
##   color  size shape
## 0   red   245     A
## 1   red   260     A
## 2   tan   189   NaN
## 3   tan   205   NaN
```

```
pd.merge(df1, df2, "right")
##   color   size shape
## 0  blue    NaN     A
## 1   red  245.0     A
## 2   red  260.0     A
## 3  grey    NaN     B
```

Joining columns (`merge` is by index)

```
df1
##   color  size
## 0   red   245
## 1   red   260
## 2   tan   189
## 3   tan   205
```

```
df2
##   color shape
## 0  blue     A
## 1   red     A
## 2  grey     B
```

```
df1.join(df2, lsuffix='_df1', rsuffix='_df2') # ~ cbind
##   color_df1  size color_df2 shape
## 0       red   245       blue     A
## 1       red   260        red     A
## 2       tan   189       grey     B
## 3       tan   205        NaN   NaN
```

```
df2.join(df1, lsuffix='_2', rsuffix='_1') # how="left"
##   color_2 shape color_1  size
## 0    blue     A     red   245
## 1     red     A     red   260
## 2    grey     B     tan   189
```

Concatenating rows

```
pd.concat([df1, df2], axis=0)
##    color   size shape
## 0    red  245.0   NaN
## 1    red  260.0   NaN
## 2    tan  189.0   NaN
## 3    tan  205.0   NaN
## 0   blue    NaN     A
## 1    red    NaN     A
## 2   grey    NaN     B
```

```
pd.concat([df1, df2], axis=1) # df1.join(df2) no suffix
##    color  size color shape
## 0    red   245  blue     A
## 1    red   260   red     A
## 2    tan   189  grey     B
## 3    tan   205   NaN   NaN
```

Aggregating

```
df = pd.DataFrame({'grp':['A','A','B','B','B','A','A'],
                   'val':[245,260,189,205,211,260,255]})
```

```
df
##   grp  val
## 0   A  245
## 1   A  260
## 2   B  189
## 3   B  205
## 4   B  211
## 5   A  260
## 6   A  255
```

```
df.groupby('grp').count()
##      val
## grp
## A      4
## B      3
```

```
df.groupby('grp').mean()
##              val
## grp
## A     255.000000
## B     201.666667
```

```
df.groupby('grp').min()
##      val
## grp
## A    245
## B    189
```

Unique values, table

```
df.grp.unique()
## array(['A', 'B'], dtype=object)
```

```
df.grp.nunique()
## 2
```

```
df.grp.value_counts() # table(df$grp) in R
## A    4
## B    3
## Name: grp, dtype: int64
```

Adding columns, apply functions

```
df = df.assign(new_col = lambda x: (x.val*1000))
```

```
df                              #  df['new_col'] = df.val*1000
##   grp  val  new_col
## 0   A  245   245000
## 1   A  260   260000
## 2   B  189   189000
## 3   B  205   205000
## 4   B  211   211000
## 5   A  260   260000
## 6   A  255   255000
```

```
df[["val","new_col"]].apply(lambda x : x/100, axis=1)
##     val  new_col
## 0  2.45   2450.0
## 1  2.60   2600.0
## 2  1.89   1890.0
## 3  2.05   2050.0
## 4  2.11   2110.0
## 5  2.60   2600.0
## 6  2.55   2550.0
```

Sorting dataframes

```
df.sort_values(by='val')
##   grp  val  new_col
## 2   B  189   189000
## 3   B  205   205000
## 4   B  211   211000
## 0   A  245   245000
## 6   A  255   255000
## 1   A  260   260000
## 5   A  260   260000
```