INFORMATIK ZUSAMMENFASSUNG:

SW	Topic	Lecture	Lab
1	Introduction	Programming languages & Python	P1
2		Variables, assignments, expressions	- " -
3		Data types, debugging	P2
4	Basics	Conditionals	- " -
5		Loops (1)	P3
6		Loops (2)	- " -
7		Functions (1)	P4
8		Functions (2)	- " -
9	Getting practical	Tuples and lists	P5
10		File I/O (input & output)	- " -
11		Dictionaries	P6 (test exam)
12		Standard- and external modules (1)	- " -
13		Standard- and external modules (2)	P7
14	FAQ	Python Patterns, Exam Preparation	- " -

Inhaltsverzeichnis

Introduction 3

V01: Introduction 3

PROGRAMMING LANGUAGES & PYTHON 3
FROM "PYTHON AS CALCULATOR" TO A PROGRAM 3

V02: Variables & Expressions 4

INTERACTIVE PROGRAMS 4
SOME THEORY: VARIABLES, EXPRESSIONS & ASSIGNMENTS 4
DEBUGGING BASICS 5

V03: Data types and Debugging 5

DATA TYPES 5
DEBUGGING WITH PDB 6
Important Python commands for data types: 7
ASCII-Code 7

Basics 8

V04: Conditionals 8

TOWARDS IMPERATIVE PROGRAMMING WITH CONDITIONAL STATEMENTS 8
FORMULATING CONDITIONS 8
WORKING WITH CONDITIONS 9

V05: Lists and Loops (1) 9

While Loop 9
Lists 9
Sequence 10
For Loop: 10
LOOPS IN-DEPTH: EXAMPLES & EXTENDED POSSIBILITIES 11

V06: Loops 12

NESTED LOOPS 12
GAME OF LIFE 13

Looping over both indexes and elements ... 14
List comprehension ... 14
V07: Functions ... 14
FUNCTION DEFINITION ... 14
DEVELOPING FUNCTIONS ... 16
MORE ON THE PASSING OF VALUES ... 16
V08: Functions ... 18
SIDE EFFECTS WHEN CALLING FUNCTIONS ... 18
RECURSIVE FUNCTIONS ... 19
Getting practical ... 20
V09: Tuples and lists ... 20
RECAP: LISTS ... 20
TUPLES ... 21
V10: File I/O ... 22
WORKING WITH FILES ... 22
FILES AND PATHS ... 23
HANDLING EXCEPTIONS ... 23
V11: Dictionaries ... 24
DICTIONARIES (ACCESSING VALUES USING A KEY) ... 25
ON OBJECTS AND METHODS ... 26

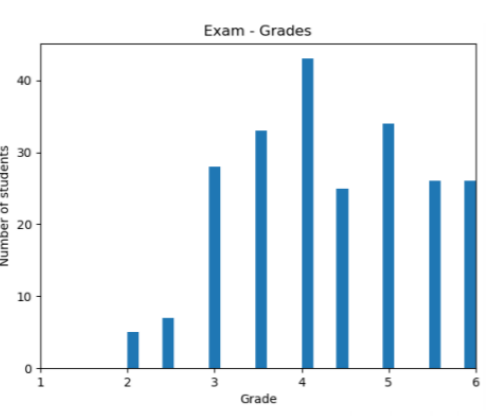
Introduction

V01: Introduction

PROGRAMMING LANGUAGES & PYTHON

Program Folie 26

```
from pylab import *
import xlrd

wb = xlrd.open_workbook("v01_grades.xls")
sheet = wb.sheet_by_index(0)

grades = []
for i in range(sheet.nrows):
    grades.append(float(sheet.cell(i, 0).value))

hist(grades, bins=30)
xlim([1, 6])
xlabel('Grade')
ylabel('Number of students')
title('Exam - Grades')
show()

failed = 0
passed = 0
for grade in grades:
    if grade < 4:
        failed += 1
    else:
        passed += 1
pie([passed, failed], labels=["passed", "failed"])
show()
```



FROM "PYTHON AS CALCULATOR" TO A PROGRAM

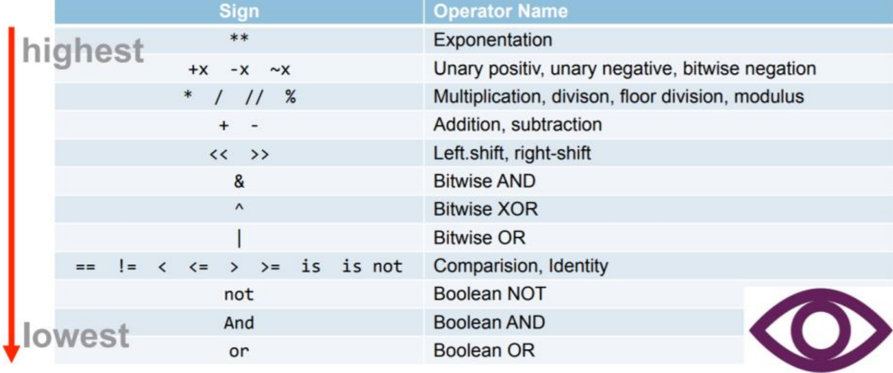
Statement ("Anweisung"):	One logical executable line of code, e.g. 1+1
Expression ("Ausdruck"):	Code snippet that yields a certain value when evaluated, e.g. 2*5 yields 10

Important arithmetic operators (Mathe):

Operation	Symbol	Example
Parenthesis	()	(2+2)*3 == 12
Power (exponentiation)	**	2**4 == 16
Multiplication	*	2*4 == 8
Division	/	13/7 == 1.85..
Modulo (remainder)	%	13%7 == 6
Addition	+	13+7 == 20
Subtraction	-	13-7 == 6

Operater in Python:

*, ** (Potenzform)	% (Gibt Restwert einer Division)
/, // (Division von floats in integers)	== macht einen Vergleich
!= ist ungleich ...	



Sign	Operator Name
**	Exponentation
+x -x ~x	Unary positiv, unary negative, bitwise negation
* / // %	Multiplication, divison, floor division, modulus
+ -	Addition, subtraction
<< >>	Left.shift, right-shift
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
== != < <= > >= is is not	Comparision, Identity
not	Boolean NOT
And	Boolean AND
or	Boolean OR

highest → lowest

Important terminology:

Argument ("Parameter"):	A value that is passed to a command in order for the command to do something with it, e.g. "Hello" in the example: print("Hello world")
String ("Zeichenkette"):	A value consisting of characters, e.g. "Hello"
Literal (the same in German):	a hard-coded value in your code, e.g. 5 or "Hello"

V02: Variables & Expressions

INTERACTIVE PROGRAMS

Aufbau Programm:

A program consists of (a sequence of) statements
- Interpreted step by step
- From top to bottom
- Empty lines are skipped
- Lines beginning with a # skipped as well
- Writing 3. and 3.0 is equivalent

input(<someString>) and int()
- Reads strings from the console
- Be careful: what is typed in by the user is interpreted as **an expression**!
- The **int()** command will convert the string returned from input to a number

Variables:

x is a variable ("Variable"):
- A **name-tag** created on the fly when using it
- Can be "filled" by **binding the name to a value** through assignment (=)
- Used anywhere where **values are expected**

SOME THEORY: VARIABLES, EXPRESSIONS & ASSIGNMENTS

Variable Assignment ("Zuweisung"):

Multi-assignment:
- <var_1> = <var_2> = ... = <expr>
  - expr's value is assigned **to all variables var_i** (➔ they all point to the same value in memory)
- <var_1>, <var_2>, ..., <var_n> = <expr_1>, <expr_2>, ..., <expr_n>
  - All expressions are evaluated, then **var_i gets assigned the value of expr_i**

Standard operator	Short-hand notation
`x = x + <expr>`	`x += <expr>`
`x = x - <expr>`	`x -= <expr>`
`x = x * <expr>`	`x *= <expr>`
`x = x / <expr>`	`x /= <expr>`
...	...

Expressions:

An expression is a construct describing a value

Forms of expressions

• **Literals** Expressions in which the value is stated directly e.g. 1.0, "Hello World"
• **Variables** References to values in memory
• **Complex expressions** Combinations with operators, e.g. 3+5
• **Calls to commands** E.g. input()
• **Any combination** of the above mentioned forms

```
c = float(input("Please state your base capital [CHF]: "))
i = float(input("Please enter an interest rate [%]: "))
print("Your return after 10 years of investment at ", i, end="")
print("% will be ", end="")  # the "," at the end of print prevents a newline

multiplier = 1 + i/100.0  # don't forget to add decimals to the divisor!

y1 = c*multiplier
y2 = y1*multiplier
y3 = y2*multiplier
...
y8 = y7*multiplier  # calculate it more easily as y8 = c * multiplier**8
print(y8, " CHF.")
```

DEBUGGING BASICS

1. Define how the program should react to different input («test cases»)
2. Find out if it does it by running it

Good to know about Python:

In Python**, everything is an object**:

- Specifically: each value (e.g. any literal) is an object
- The term "object" is best understood as a unique representation of that value in memory
- Objects of certain types (e.g. float, integer, string, Boolean) are immutable
  - they never change their value; instead, new objects are created

Differences between Python and other languages:

- In other programming languages variables are like boxes (locations in memory):
  - Assignment puts a value in the box (a=1); assigning one variable to another duplicates the value
- Variables are names (or name tags) in Python:
  - Assignment attaches the **name tag to an individual object (a=2);** assigning one variable to another (b=a) adds the b name tag to the same object that already has the a name tag

Python is **dynamically typed**

- Variables don't have to be defined; they are **created on the fly when used**
- Variables can be bound **to any type of value** (because of them being name-tags for objects)
  - the type of a variable is dynamically determined at runtime

V03: Data types and Debugging

DATA TYPES

Overview of Data Types:

- Numbers:
  - int (42), float (42.0)
- Boolean:
  - bool (truth values: either True or False)
- Strings:
  - str
    - "This is a string"; 'This is also a string'
    - '42'; "42.0"
- Collections:
  - tuple
    - (42, 42.0)
  - list
    - [42, 42.0, "This is a String"]
  - dictionary (→ see V05/V09 and V11)
    - {"first": 42, "second": 42.0}

Immutable	Mutable
bool	
int	
float	
str	
complex	
tuple	list
	dict
frozenset	set
bytes	bytearray
NoneType	

Dynamic Typing

- We do not have to define the type of a variable
- **Finding out** about the **type of a variable**

```
>>> x = 42.0
>>> type(x)
float
```

- This **type is not fixed** and can **change at runtime** → called **dynamic typing**

```
>>> x = 42.0
>>> type(x)
float
>>> x = 'fortytwo'
>>> type(x)
str
```

Float

Decimal numbers are stored as type float are represented with **limited precision**

```
>>> 0.1 + 0.2
0.30000000000000004
```

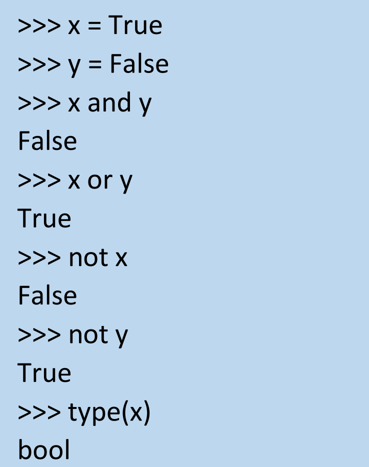
- int / int = float

```
>>> type(2/1)
float
>>> type(2/1.0)
float
```

Boolean and Relational Operators

True or **False**

- often **the result of relational operators** (comparisons, e.g. >, ==, !=, (xor)



```
>>> x = True
>>> y = False
>>> x and y
False
>>> x or y
True
>>> not x
False
>>> not y
True
>>> type(x)
bool
```

x	y	x and y	x or y	x ^ y	not x
T	T	T	T	F	F
T	F	F	T	T	F
F	T	F	T	T	T
F	F	F	F	F	T

String:

• string is a **sequence of characters** in double or single quotes: 'Hello'

String operations

help(str)

• **Concatenation** string1 + string2:	'Hello' + ' World' → 'Hello World'
• **Repeat string** string1 * n:	'.' * 5 → '.....'
• **Get character at index 5** (0-based!) string1[5]:	'abcdefgh'[5] → 'f'
• **Get substring with slicing** string1[start:stop]:	'abcdefgh'[:5] → 'abcde'
• **Get length of a string** len(string1):	len('Hello') → 5
• **Find patterns** pat in string1:	"an" in "apples,bananas,tomatoes“ → True
• **Split a string** string.split(delimiter) in a List:	"apples,bananas,tomatoes".split(',') → ['apples', 'bananas', 'tomatoes']

DEBUGGING WITH PDB

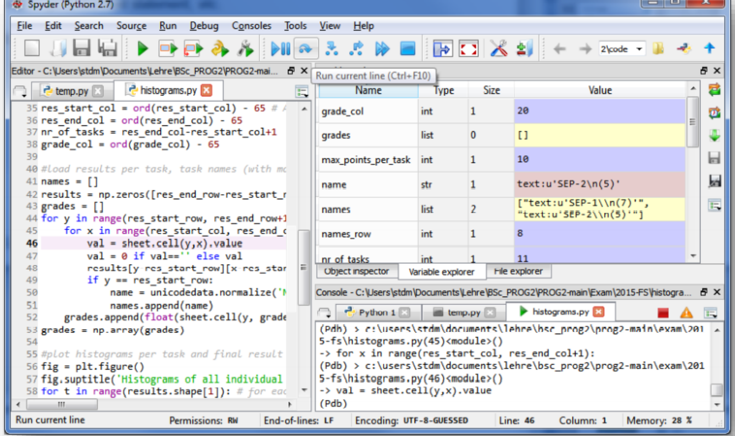
WHEN A PROGRAM DOESN'T DO WHAT YOU WANT IT TO:

1. Read, understand and verify the code with test cases
2. Use print output to monitor key values
3. Use a debugger

PDB in Spyder

Run a script via the blue debug buttons

- Execute current line, jump in and out of commands/calls, resume normal execution
- Current line will be lightly highlighted (with the line number in **bold face** font)
- Use the variable explorer to examine variable values during steps
- Set break points ● via double click on line number
  - Pressing will run the program up to (not including) the line of the next breakpoint



Important Python commands for data types:

type() and the type conversion functions int(), float(), str(), bool() etc.

```
>>> ord('A')
65
>>> chr(65)
'A'
>>> chr(ord('A') + 3)
'D'
>>> ord('1')
49
>>> hex(ord('1'))
'0x31'
```

ASCII-Code

Example: «A» has ASCII-code $41_{16} = 65_{10}$

HEX-CODE	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1...	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2...	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6...	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	Q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

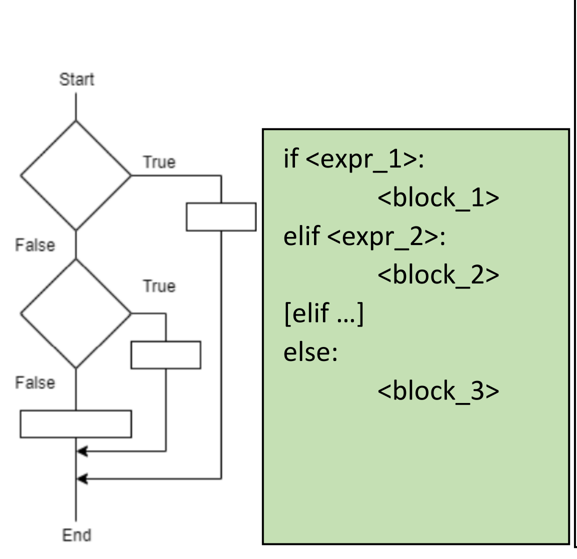
www.VirtualUniversity.ch

Basics

V04: Conditionals

TOWARDS IMPERATIVE PROGRAMMING WITH CONDITIONAL STATEMENTS

The "if" family of statements:




```
Example:
a = b = 2
c = False
if not c:
        if b < a:
                b += 5
                a = b-1
        elif a < b:
                c = True
        else:
                if a+b < 4:
                        c = False
                a = 11
                b = 2.2
print(a, b, c)
```

FORMULATING CONDITIONS

Boolean expressions:

- **False:** False, 0, the empty string (''), other empty "containers" (later more on them)
- **True:** All other values
- Attention: "evaluate to True/False" does not mean "equality to True/False"!

Logical operators	Code (x=True; y=False)
Negation	`not x # False`
Conjunction	`x and y # False`
Disjunction	`x or y # True`
Precedence	`not > and > or`

Relational operators	Code (x=5; y=7)
Less than Less than or equal to	`x < y # True` `x <= y # True`
Greater than Greater than or equal to	`x > y # False` `x >= y # False`
Equal to	`x == y # False`
Not equal to	`x != y # True`

```
Example:
print(3 > 2) → # True
print((2*3) + 4 != 2*3 + 4) → # False
print(True or False) → # True
print(7 or 0) → # 7
print(True and "OK" or "KO") → # OK
print(False and "OK" or "KO") → # KO
```

Observations
- and and or perform Boolean logic, but...
- They do not return Boolean values

*The special behaviour of **and** and **or***

- and and or perform Boolean logic as expected, but...
- they return one of the actual values they are comparing

Examples:
- '' and 'b' → # == '' (and returns the first False value)
- 'a' and 'b' → # == 'b' (if all values are True, and returns the last one)
- 'a' or 'b' → # == 'a' (or returns the first True value)

- '' or [] or 0 → # == 0 (if all values are False, or returns the last one)
- 'a' and 'b' and '' → # == ''
- 'a' or <someCode> → # == 'a' (someCode is never executed / evaluated

```
print("Please think of a number from 1 to 10. Ready?")
res = input()

greater = "Is the number greater than"
equal = "Is the number equal to"

print(greater, 5, "?")
res = input()
if res == "yes":  # 6<=n<=10
    print(greater, 7, "?")
    res = input()
    if res == "yes":  # 8<=n<=10
        print(greater, 9, "?")
        res = input()
        if res == "yes":  # n=10
            n = 10
        else:  # 8<=n<=9
            print(equal, 8, "?")
            res = input()
            n = 8 if res == "yes" else 9
    else:  # 6<=n<=7
        print(equal, 6, "?")
        res = input()
        n = 6 if res == "yes" else 7
```

```
# ...
else:  # 1<=n<=5
    print(greater, 2,
    res = input()
    if res == "yes":
        print(greater
        res = input()
        if res == "ye
            n = 5
        else:  # 3<=n
            print(equ
            res = inp
            n = 3 if
    else:  # 1<=n<=2
        print(equal,
        res = input()
        n = 1 if res

print("You choose num
```

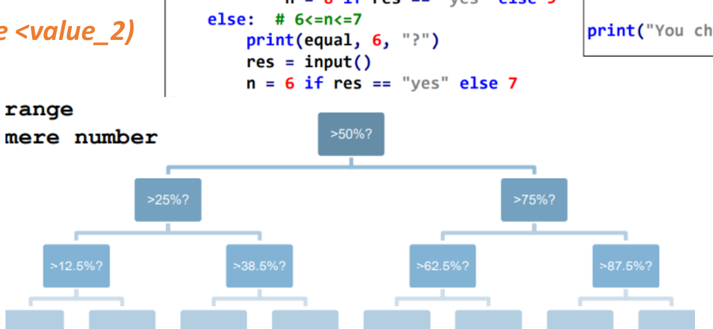
WORKING WITH CONDITIONS

```
>>> a = 3 if 5 > 2 else 4
>>> a
3
>>> a = 3 if 5 < 2 else 4
>>> a
4
```

- ***if 0 < x < 10*** is identical to ***if (0 < x) and (x < 10)***
- ***<some_variable> = (<value_1> if <expression> else <value_2)***

«If»-constructs

- There's a clear structure in the task of splitting up the search space:
  - Partition current range into halves
  - Let user choose lower or upper half
  - Chosen half becomes new current range
  - Repeat until current range is a mere number
  - Return chosen number



V05: Lists and Loops (1)

While Loop

- while <expr>:
  <block>
- First, the expr gets evaluated

If False: continue program after loop
(next statement with same indent as while)
print in this sample

If True: execute statements of block,
then go back to while <expr>

```
Example:
a = 8
b = 1
while a > 1:
        b += 3
        a = a / 2
print(a, b)
```

a > 1	a	b
	8	1
True	4.0	4
True	2.0	7
True	1.0	10
False		

Lists

- Write a program where a user can enter a list of names/dates etc. After the user has finished entering names/dates..., your program prints all entered values
- Fortunately, the data type **list can be used to store arbitray many values**. Lists contain elements. In Python, you **can store all data types** in a list.

Using Lists

Lists can be created in different ways:

```
first_list = [] # creates an empty list (without any values)
second_list = [1, 2, 3, 4] # creates a list with the elements 1,2,3 and 4
third_list = ["abc", "def"] # creates a list with the elements "abc" and "def"
print(third_list) # prints ["abc", "def"]
```

list.append()

append adds to the end the list:

```
first_list = ["Steve McQueen"] # creates a list with one value
first_list.append("Peter Fonda") # Adds the string "Peter Fonda" to the list
print(first_list) # prints ["Steve McQueen", "Peter Fonda"]
first_list.append("Paul Newman") # Adds the string "Paul Newman" to the list
print(first_list) # prints ["Steve McQueen", "Peter Fonda", "Paul Newman"]
```

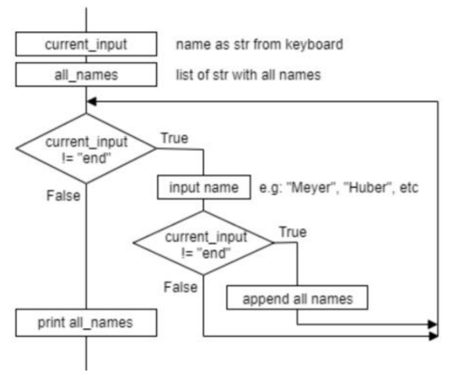
len(), list[], del()

```
second_list = [1, 2, 3, 4] # creates a list with the values 1,2,3 and 4
print(len(second_list)) # prints the length of the list (number of values) → in this case 4
print(second_list[0]) # prints the first value of the list, in this case 1
del(second_list[1]) # deletes the first element (note that we start counting with zero)
print(second_list) # prints [1,3,4]
```

index	0	1	2
second_list	1	3	4

Example The "user-input" program:

A user can enter a list of names. When the user wants to finish entering names "end" has to be typed. Your program prints all entered values (without "end").

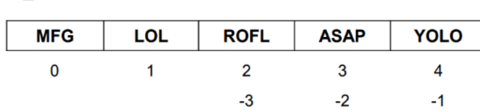



```
current_input = ""
all_names = []                                  # 1

while current_input != "end":                   # 2
    current_input= input("Please enter a name")
    if current_input != "end":                  # 3
        all_names.append(current_input)


print (all_names)                               # 4
```

Sequence

Lists

The **list data type is a so called sequence**. A sequence is **an ordered set**. As we have seen in our list, all elements in the list are ordered:



a_list = ['MFG', 'LOL', 'ROFL', 'ASAP', 'YOLO']

MFG	LOL	ROFL	ASAP	YOLO
0	1	2	3	4
		-3	-2	-1

Stings

is a **sequence of characters** in double or single quotes: 'Hello'

H	e	l	l	o

- **text[<start>:<end>]** accesses all characters between **index start and end (excluding end)** in the string text
- A negative end **counts backwards from text's last character**

➔ 'Holladihuuu'[0:-2] is 'Holladihu'

H	o	l	l	a	d	i	h	u	u	u
0	1	2	3	4	5	6	...	-3	-2	-1

range()

range() is a function ("Funktion"): a block of code that is defined somewhere else.

```
list(range(<end_value>)) # returns a list of integers from 0 to end_value-1
list(range(5)) # returns [0, 1, 2, 3, 4]
list(range(<start_value>, <end_value>[, <step>])) # SYNTAX
# returns a list (sequence) of integers including the start_value and
# excluding the end_value incrementing by step
range(1, 5) # returns [1, 2, 3, 4]
Creating ranges with steps:
list(range(0, 10, 2)) # returns [0, 2, 4, 6, 8]
list(range(10, 0, -2)) # returns [10, 8, 6, 4, 2]
list(range(0, -10, -1)) # ret. [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

For Loop:

```
for <item> in <sequence>:
        <block>
```

- for-loops are used when you want **to iterate over some sequence**

➔ E.g. a range of numbers, characters in a string, or elements in a list

➔ The precise number of runs is known in advance (e.g., length of the sequence)

Python's "for each" loop can be used as an inductive loop ("Zählschleifen")

- for i in range(<total_count>):
  <block>
- Python **executes block exactly total_count times**
- **i is a integer** variable ("counter variable") running from **0 to total_count-1**

```
# Task: write a certain text 10 times
for i in range(10):
        print("I will not come late to class.")
```

LOOPS IN-DEPTH: EXAMPLES & EXTENDED POSSIBILITIES

For Loop Example:

Goal: Find the greatest product of five consecutive digits in this 1000-digit number:

```
number = 731671765313 … 420752963450 # total 1000 digits
str_number = str(number) # 7 * 3 * 1 * 6 * 7 = 882
largest_product = 0 # to remember the largest product so far …
at_position = 0 # … and where it occurred
for i in range(len(str_number) - 5):
product = int(str_number[i]) * int(str_number[i+1]) * etc.
if product > largest_product:
largest_product = product
at_position = i
print(largest_product, "starting at position", at_position)
```

```
for j in range(5):
        product *= int(str_number[i + j])
```

Find the difference between the sum of the squares of the first one hundred natural numbers and the square of their sum.

```
sum_of_squares = 0
sum_of_numbers = 0
for number in range(1, 101):
        sum_of_squares += number**2
        sum_of_numbers += number
print(sum_of_numbers**2 - sum_of_squares)
```

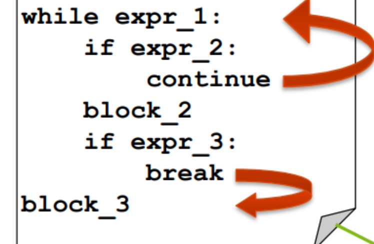
While continue and brake

break:

- exits the current loop immediately without evaluating the condition(the next statement evaluated is block_3)

continue:

- skips the remainder of the current iteration (it re-enters the loop by evaluating expr_1 and continues from there)



A complicated nested ("verschachtelte") structure…

```
while expr_1 and not expr_2:
    if expr_3:
        x = calculate_something()
        if expr_4:
            y = complicated_code(x)
            if expr_5:
                real_cause_part_1()
                if not already_finished():
                    real_cause_part_2()
                else:
                    expr_2 = True
```

…can be thus transformed:

```
while expr_1:
    if not expr_3: continue
    x = caluculate_something()
    if not expr_4: continue
    y = complicated_code(x)
    if not expr_5: continue
    real_cause_part_1()
    if already_finished(): break
    real_cause_part_2()
```

Example:

```
attempt_count = 0
while True:
	i = input("Please enter a number between 1 and 5: ")
	if i in ['1', '2', '3', '4', '5']:
		i = int(i)
		break
	else:
		if attempt_count > 3:
			print("1, 2, 3, 4 or 5, please... not", i)
		elif attempt_count > 2:
			print("Did you have your coffee yet?")
			print(i, "is not a digit between 1 and 5.")
		attempt_count += 1
if attempt_count > 0:
print("Thank you!")
print("You chose number", i)
```

```
if count > 3:
    print "..."
elif count > 2:
    print "..."
```

- If the order was reversed, elif count > 3 would never be evaluated(because of elif → clarify this for yourself!)
- If elif was replaced by if, both would get evaluated

V06: Loops

NESTED LOOPS

<table>
<tr><td>Iterating over items in lists: "for each"</td><td>Iterating over item positions in lists: «inductive loop»</td></tr>
<tr><td>weekdays = ['Mon', 'Tues', 'Fri']
for day in weekdays:
    print("Today is a...", sep="")
    print(day)</td><td>weekdays = ['Mon', 'Tues', 'Fri']
for i in range(len(weekdays)):
    print("Today is a...", sep="")
    print(weekdays[i])</td></tr>
<tr><td colspan="2">>>>
Today is a... Mon
Today is a... Tues
Today is a... Fri
>>></td></tr>
<tr><td>- Item can be used directly
- Applied if every item in a list needs to be manipulated in the block</td><td>- Item is accessed over its position in the list (grab the item at position i)
- Applied for access on more than one item in the block</td></tr>
<tr><td></td><td>numbers = [5, 3, 1, -1, -3]
for i in range(len(numbers))[1:]: # range(1, 5)
    difference = numbers[i] - numbers[i-1]
    if difference != 2:
        print("Difference not 2")</td></tr>
</table>

- Order of loops changes order of operations
  - For each item of the outer loop, execute the inner loop
  - For each tuple (outer_item, inner_item),execute the code in the inner loop
  - Common examples of loop nesting
- x/y coordinates for printing something on the screen
- x/y coordinates of cells in a table/grid (mathematically: a matrix)
- Best practice: row-wise processing

```
for y in rows:
        for x in columns:
        # do_something_at_coordinate(x, y)
for row in rows:
        for col in columns:
        # do_something_at_cell(col, row)
```

Iterating over two things in a nested way:

```
fruits = ["apple", "banana", "melon"]
for i in range(2, 6, 2):
    for f in fruits:
        print(str(i) + " " + f + "s")
```

```
>>>
2 apples
2 bananas
2 melons
4 apples
4 bananas
4 melons
>>>
```

```
fruits = ["apple", "banana", "melon"]
for f in fruits:
    for i in range(2, 6, 2):
        print(str(i) + " " + f + "s")
```

```
>>>
2 apples
4 apples
2 bananas
4 bananas
2 melons
4 melons
>>>
```

Nested lists

x = [0, 1, 2, 3, 4, 5] # is called a list of ints

- It’s the same as x = range(6)
- type(x): <type 'list'> → list is a data type like int, float, …
- x[1]: 1 → provides index access just as with strings
- [] serves to goals in python: index access (already seen) and list construction (new here)

index	0	1	2	3	4	5
x	0	1	2	3	4	5

x = [[0, 1], [2, 3], [4, 5]] # is called a list of lists of ints

- The data type of the elements of a list can be any type (even a list again)
- x[2][1]: 5 → index access works for nested layers (x[row][column])

x[][]	x[…][0]	x[…][1]
x[0][…]	0	1
x[1][…]	2	3
x[2][…]	4	5

GAME OF LIFE

- Zero player game: An initial on/off configuration of"cells" on a grid evolves according to 3 simple rules
- Can be seen as a simulation of life itself
- Background:
  - Created in 1970 by mathematician John Conway (*1937, † 2020)
  - Technically a general computer («Universal Turing Machine»)and instance of a «cellular automaton»
- How to play:
  - Beginners: Watch the fascinating evolution of emerging structures from random configurations
  - Advanced: Create initial configurations that generate patterns with certain properties (see Wikipedia)
  - Interview with John by Numberphile: https://www.youtube.com/watch?v=R9Plq-D1gEk
- The rules:
  - Any alive cell with <2 alive neighbours dies → "under-population"
  - Any alive cell with >3 alive neighbours dies → "overcrowding"
  - Any dead cell with exactly 3 alive neighbours becomes alive → "reproduction"
  - (Any alive cell with 2 or 3 alive neighbours lives on)

```
import os  # for clear screen
import copy  # for duplicating arrays
import random  # for random number generator
import colorama  # to address coordinates on screen

# definition of grid size
rows = 20
cols = 79

# initializations, e.g. board with random 0s and 1s
random.seed()
colorama.init()
generation = 0
empty = False

board = [[int(random.choice('01'))
        for x in range(cols)] for y in
range(rows)]
os.system('cls' if os.name == 'nt' else 'clear')

while not empty:
    generation += 1
    new_board = copy.deepcopy(board)
    for y in range(rows):
        for x in range(cols):
            # count the number of alive neighbours
            # evaluate Conway's rules
            # print updates directly to console
    board = copy.deepcopy(new_board)
```

- **board** (the 2D list of cells) is filled at random using a shorthand **for**-loop notation (→ **list comprehension**, see appendix)
- Outer loop runs infinitely for now
  → Care to compute **empty** later
- The state of the new board is based on its *entire* previous state
  → Work with copy of the **board** to not override previous statuses prematurely (see V08)
- 2 **for** loops are prepare to visit each cell and…
- All «real work» is just plain text comments for now
  → Care to design the needed code later
- Prepare next run by copying back the current **new_board** to the new **board**

Looping over both indexes and elements

```
fruits = ["apple", "banana", "melon"]
for i, item in enumerate(fruits):
        print(i, item)
```

```
>>>
0 apple
1 banana
2 melon
>>>
```

List comprehension

```
squares = [num*num for num in range(21) if num%2 == 0]
```

```
[ <expression> for <item> in <list> if <conditional> ]
```

is equivalent to

```
for <item> in <list>:
    if <conditional>:
        <expression>
```

V07: Functions

FUNCTION DEFINITION

def <valid_name>([<param_1> [, <param_2> [, …]]]):
    ['''<docstring>''']
    <block>
<some_variable> = <function_name>(<param_1>, …, <param_n>)

- The def keyword introduces the function header (or signature / "Kopf")
- The function name valid_name
  - Same rules apply as for variable names
- A pair of parenthesis () and a colon :
  - The parenthesis indicate a function (as opposed to a variable) and are mandatory in any case
  - The optional parameters param_i receive values from outside and serve as variables inside
  - The colon serves as in for/while/if as a separator between header and body

- The function body ("Rumpf") block
  - Is indented; may contain zero or more return statements
  - May contain any code and control structures you already know

```
def get_braking_distance(v0):
        '''
        Calculates braking distance [m]
        with mu = 0.3 and v0 [m/s]
        '''
        mu = 0.3 #Coefficient of friction
        g = 9.81 # gravitational acceleration
        return 0.5*v0**2 / (mu*g)

velocity = 30 #m/s
print ("The braking distance for v0=", velocity, "is",get_braking_distance(velocity), "m")
```

- **0, 1 or more parameters** are possible
  - just list them in the function definition
- **0, 1 or more return statements** are ok
  - the function immediately ends there, the control flow returns to after the call
- Self-made functions **can call other functions**

```
def hello1():
        print ("Hello, world")
def hello2(name):
        print ("Hello,", name)
        return
def hello3(name, count):
        for i in range(count):
                hello2(name)
def is_even(a):
        '''returns True if the given integer
        a is an even number, else False'''
        if a % 2 == 0:
                return True
        else:
                return False
```

```
>>> hello1()
Hello, world
>>> hello2('Thilo')
Hello, Thilo
>>> x = 1
>>> hello3('Thilo', x if is_even(x) else x+1)
Hello, Thilo
Hello, Thilo
>>>
```

Global and local variables

- A global variable can be used in functions on the right hand side of an assignment (e.g. b above)
- However, if it is tried to reassign it (use on the left hand side), …
  - A new local variable of equal name is created instead (e.g. a above)
  - This new local variable a shadows ("verdeckt") the global variable a
  - The global a is no longer visible in function()

```
a, b = 'one', 'two'
print ("outer a,b=", a, b)
def function():
        #a, b = 1, 2
        a = 1
        print ("inner a,b=", a, b)
function()
print ("outer a,b=", a,b)
```

```
>>>
outer a,b= one two
inner a,b= 1 two
outer a,b= one two
>>>
```

DEVELOPING FUNCTIONS

• Give each function exactly one responsibility (functionality)
(i.e., computing a result, but not caring for user interaction like raw_input / print)
• Write functions so that they can be generally used
(i.e., not just in the case you currently have in mind)
• Encapsulate the code in the function from the rest of your program
(i.e., only communicate with a function via parameters and return value)

```
def create_triangle(character, num_lines):
    '''Produce a string containing a triangle of a certain number of lines
      in height, using the certain character as the drawing element.'''

    #initialise the triangle that we later want to return
    pattern = ''

    #first loop: include middle element here
    for y in range(int(num_lines/2) +1):
        pattern = pattern + (y+1)*character + '\n' #append rows

    for y in range(int(num_lines/2)):
        pattern = pattern + (int(num_lines/2) - y)*character + '\n'

    return pattern

figure = create_triangle('*', 5)
print (figure)
```

Not cared for yet: Ensuring uneven number of lines

Provides the variable that contains the string that we subsequently build in an iterative fashion (add to it until its finished)

Creates the rows up to (and including) the middle element.
Challenge 1: Correct range to cover exactly these rows
Challenge 2: Find expression that produces correct number of characters per row, only depending on the current loop counter

Calling functions

Basic syntax (eigene Funktion)

- <some_variable> = <function_name>(<param_1>, …, <param_n>)

Dotted (i.e., object-oriented) syntax (Methodes)

- <some_variable> = <my_variable>.<function_name>(<param_1>,…)

MORE ON THE PASSING OF VALUES

The return statement

• The return statement stops the execution of the function and returns a value
• A return statement can occur anywhere in the function definition
• Several values can be returned at once: return (<val_1>, <val_2> [, …])

```
def function(a):
        if a:
                return
        else:
                return (a, not a)
x = function(True)
y = function(False)
z1, z2 = function(False)
print ("x: ", x, type(x))
print ("y: ", y, type(y))
print ("z1, z2: ", z1, type(z1), z2, type(z2))
```

```
>>>
 x: None <class 'NoneType'>
 y: (False, True) <class 'tuple'>
 z1, z2: False <class 'bool'> True <class 'bool'>
>>>
```

The special value None

None is a special object in Python indicating «no value at all»
• Compare a variable to None only by using the is operator: if not x is None: print x
Every function in Python returns something
• No return statement means actually return None at the end
Just return without any value is equivalent to return None
• Don't do this – it's bad style

Everything is an object in Python

- A notion of some unique «entity» (Nothing more to; doesn't mean «Object Oriented Programming»)
- Features of any object (Attention – names are no feature of objects!)

Features	Explanation
identity (=location)	Some identifier that makes sure we can tell if two names refer to the same object or not → actually it's memory address, e.g., `id(2) = 31370940`
value	e.g., `2`
type	e.g., `<type 'int'>`

- A variable is a name that references (is bound to) an object

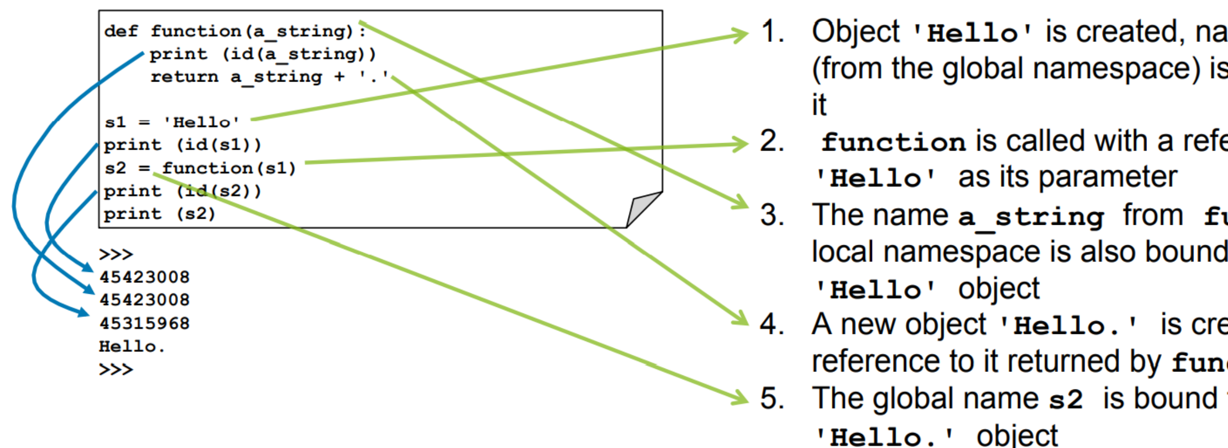
Examples of Python objects:

- The natural numbers like 1, 2, 3
- Every other literal you place in your code, e.g. 'Hello, world'
- Every function, keyword, type, ... everything

What is actually passed between caller and function?

1. A reference to the parameter object is passed from the caller to the function
2. There, the parameter name inside the function is (additionally) bound to that same object
3. This is called call by reference

Aufbau einer Funktion:



Functions as Objects

Recall: In Python, everything is an object

- Functions are objects as well
- The function definition is nothing but the binding of a name to a value(function name to its parameter list and body)
- Functions (because they are objects) can be assigned to other variables

```
def approx_sqrt(a):
    x = 1.
    for i in range(9):
        x = x - (x**2 - a) / (2*x)
    return x
```

```
>>> f = approx_sqrt
>>> f(x)
1.4142135623730951
>>>
```

```
>>> x = 2
>>> x
2
```

Import math

```
import math

def df(f, x, h):
    '''Computes the numerical approximation of the derivative of f at point x:
       f\'(x), using a small increment h around the value x'''
    return (f(x+h) - f(x-h)) / (2.0*h)

f = math.sin
for x in [0.0, 0.5*math.pi, math.pi]: #generate some values in [0, pi]
    dx = df(f, x, 0.000001) #compute the numerical derivative of sin(x)
    print ("f'(",x,")=", dx, " should be: ", math.cos(x)) #check if it's near the expected result, cos(x)
```

V08: Functions

SIDE EFFECTS WHEN CALLING FUNCTIONS

Strictly speaking: **All other effects of a function call besides the return value**
Practical definition: Changes to values other than
the return value that are observable from the outside

- Change of any value in the scope of the calling block
- Example: **Change of a global variable**

```
# global state variable
state = [1]

# some function definition
def increase_state():
    state[0] = state[0] + 1

# main program
result =
do_something_significant()
if result:
    increase_state()
# ...
```

→

```
# global state variable
state = 1

# some function definition
def increase_state(current_state):
    new_state = current_state + 1
    # maybe do something additional… e.g.,
    # log to a file
    return new_state

# main program
result = do_something_significant()
if result:
    state = increase_state(state)
# ...
```

Intended side effects

```
def increase(my_list, factor):
    ''' Increase each element in my_list (a list of numbers) by
    factor. Attention: Modifies the elements in the original
    object! '''
    for i in range(len(my_list)):
        my_list[i] = my_list[i] * factor
dozen = list(range(1, 13)) # 1..12
print(dozen)
increase(dozen, 2) # 2..24
print(dozen)
```

```
>>>
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24]
>>>
```

```
def increase(my_list, factor=2, method="mult"):
    ''' Increase each element in my_list (a list of numbers):
        If method == "add", the factor is added to each element.
        If method == "exp", each element is raised to the power of factor.
        If method has any other value, each element is multiplied by factor.
        Attention: Modifies the elements in the original object! '''
    for i in range(len(my_list)):
        if method == "add": my_list[i] = my_list[i] + factor
        elif method == "exp": my_list[i] = my_list[i] ** factor
        else: my_list[i] = my_list[i] * factor

dozen = list(range(1, 13))  # 1..12
print(dozen)
increase(dozen)  # 2..24 (*2)
print(dozen)
increase(dozen, method="add")  # 4..26 (+2)
print(dozen)
increase(dozen, 3, "exp")  # 64..17576 (**3)
print(dozen)
```

Fill up from behind: If any parameter has a default value, all following parameters also need to have one.

Mapping of given- to expected parameters: Arguments in a function call are mapped to the expected parameters in the order of their appearance; missing arguments *at the end* will be filled up with default values (if existent; otherwise, an error is raised).

Named parameters: If you just want to specify a few out of many parameters, you can.

Overriding default values: The default values are overridden by just using the function as if they didn't exist.

Unintended side effects

- Avoid side effects whenever possible
- If a function has side effects – don't return a value (to indicate that the result is communicated otherwise)

```
def problematic_function(my_list=[]):
    my_list.append(1)
    return my_list
```

```
>>> problematic_function()
[1]
>>> problematic_function()
[1, 1]
>>> problematic_function([])
[1]
>>>
```

RECURSIVE FUNCTIONS

- A recursive function is one that calls itself to fulfil its purpose

# iterative pseudo code	# recursive pseudo code

def find_innermost(puppet):
    while can_be_opened(puppet):
        puppet = open(puppet)
    return puppet
inner = find_innermost(matrjoschka)
	def find_innermost(puppet):
	    if not can_be_opened(puppet):
	        return puppet
	    else:
	        next_puppet = open(puppet)
	        return find_innermost(next_puppet)
	inner = find_innermost(matrjoschka)

def loop_power(a, n):
    ''' Raises a to the power of n using a simpleloop '''
    power = 1
    for i in range(n):
        power *= a
    return power
	def recursion_power(a, n):
	    ''' Raises a to the power of n using recursion.
	    Idea: a^n = a^(n/2)*a^(n/2) (if n is even)
	    a^n = a^(n/2)*a^(n/2)*a (otherwise)
	    This is very fast because of less mult! '''
	    if n == 0:
	        return 1
	    else:
	        if n % 2 != 0:
	            return a * recursion_power(a, n-1)
	        else:
	            half_power = recursion_power(a, n/2)
	            return half_power * half_power

➢ Loop version has linear runtime a 1'000'000 needs 1'000'000 multiplications ➢ It's quite difficult to formulate the algorithm to the right using iteration	➢ Recursive version has logarithmic runtime a 1'000'000 needs only 26 multiplications

Important aspects of any recursive function definition

```
def factorial(n):
    ''' Returns n! for any positive integer n '''
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Termination criterion: The base case

Recursive call: Solve smaller problem

If-Else statement: To tell the two apart

Easy: The base case

Spare work for later: The recursive call

Easily read, sometimes difficult to formulate: Current work

Use cases

Steps to writing a recursive function

1. Write the **function head (name & parameters)** for the recursive function
2. **Write a doc string** that describes what the function does
3. Determine **the base case** (there may be more than one), and its solution(s)
4. Determine **what smaller problem** (or problems) to solve

ASSUME **the recursive call works**, i.e., that it will correctly compute the answer

5. Use the **solutions of the smaller problem to solve the larger problem**

FUNCTIONS INSIDE OF FUNCTIONS

Functions can be called anywhere an expression is expected!

Getting practical

V09: Tuples and lists

RECAP: LISTS

Import random (random.randint())

Computer, tell me where should I go eat:

```
import random
# List of restaurants
restaurants = ['Mensa', 'Royal Mangal City', 'Migros', 'Tibits']
# Computer, choose one
which_one = random.randint(0, len(restaurants)-1)
# Let me know where I eat today
print("Why don't you go to", restaurants[which_one], "today?")
```

Lists usage summary

Create a list	**Access members of a list using slicing**
• x = [] # empty list • x = [1,2,3,4] # list with known elements • x = [0]*5 # copy a zero 5 times • x = range(5) # creates [0,1,2,3,4]	• x[start:end] # items start through end-1 • x[start:] # items start through the rest • x[:end] # items from 0 to end-1 • x[:] # a copy of the whole list • x[start:end:step] # from start # through not past end, # by step • x[-1] # last item in the list • x[-2:] # last two items in the list • x[:-2] # everything except the last 2 • x[::-1] # to reverse the list
Length of a list • len(x) # returns the length of x	
Append and delete items • x.append(5) # appends a five to the end • del(x[4]) # removes the 5th item	
Changes items • x[2] = 4 # gives the 3nd item value 4	
Loop over lists • for i in range(len(x)): print(x[i]) # indirect iteraton • for item in x: print(item) # direct iteration	

Array

• You can build a list of lists → x = [[1,2], [3,4]]

Import copy (copy.deepcopy())

```
import copy
# List of restaurants
my_restaurants = ['Mensa', 'Royal Mangal City', 'Migros', 'Tibits']
your_restaurants = copy.deepcopy(my_restaurants) # ==
your_restaurants = my_restaurants[:]
your_restaurants[-1] = 'Bloom'
# Print our restaurants
print ("These are MY favourite restaurants")
for rest in my_restaurants:
        print("\t" + rest)
print ("These are YOUR favourite restaurants")
for rest in your_restaurants:
        print("\t" + rest)
```

```
These are MY favourite restaurants
        Mensa
        Royal Mangal City
        Migros
        Tibits
These are YOUR favourite restaurants
        Mensa
        Royal Mangal City
        Migros
        Bloom
```

```
# Read in Restaurants until empty line
restaurants = []
while True:
    restaurant = input("Another favourite Restaurant: ")
    if restaurant == "":
        break
    else:
        restaurants.append(restaurant)

# Choose one randomly
import random
selected = random.choice(restaurants)  # mind the easy access to a random in the list

# Print all restaurants and mark selected
print()
for item in restaurants:
    if item == selected:
        print("->",item)
    else:
        print("  ",item)
```

```
>>>
   Mensa
-> Royal Mangal City
   Migros
   Tibits
>>>
```


TUPLES

Tuples usage summary "Tuples are read-only lists"

In general:Tuples are like lists but **tuples are immutable** (→ cannot be changed) ➢ x = (1,2) x[1] = 10 # does not work	**Why use tuples?** ➢ Tuples are often used when more than one value is returned from a function ➢ With tuples you can do neat value switching in one line of code

Create tuples
x = (1,2) # comma is the construction operator
x = 1, 2 # parentheses are for readability only
x = (1,) # and also indicate the data type tuples when printed
	Pack variables in to tuples
	x, y = 2, 3
	z = (x, y)

Length of a tuple len(x)	**Unpack tuples into variables** x, y = z # same x = 2, y = 3 y, x = z # switched x = 3, y = 2
• How do we add / remove items? → Create new ones with slicing and + operator	

Tuples in functions (min and max)

```
def get_min_and_max(values):
        return min(values), max(values) # Tuple packing
temperatures_this_week = [22,20,19,19,18,17,20]
min_and_max = get_min_and_max(temperatures_this_week)
min_temp, max_temp = min_and_max # Tuple unpacking
print ("This week we expect a low of", min_temp, "and a high of", max_temp)
```

Developing functions

What does the header look like?

- ➢ Name of function find_closest_point
- ➢ How many parameters and how to name them: 2: all_points, p

Does the function return a value (multiple values)?

Yes, 2: a tupel(x,y) of coordinates of closest point as well as distance

How does a function call look like?

closest_p, dist = find_closest_point([(1,2),(0,0),(-1,-1)], (1,1))

```
import math
def find_closest_point(all_points, p):
        xp, yp = p # unpacking tuple
        min_dist = 1e10 # something large
        for xi, yi in all_points: # unpacking tuple
                dist = math.sqrt((xp-xi)**2 + (yp-yi)**2)
                if dist < min_dist:
                        min_dist = dist
                        min_p = (xi,yi) # packing tuple
        return min_p, min_dist
```

V10: File I/O

WORKING WITH FILES

Workflow in Python

1) <file_object> = open(<filename>, <mode>)

- opens a file for reading
- returns a "file object"
- Modes (a string): read or append or write; add + for both r/w; optionally add binary

2a) <some_string> = <file_object>.readline()

- reads one line from file object f and stores it in some_string
- the empty string ('') indicates the end of file ("EOF")

2b) <file_object>.write(<some_string>)

- Writes some_string to file file_object

3) <file_object>.close()

- closes file object file_object
- (close a file ➔ no further file operations possible)

mode
Read Only ('r') : Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exists, raises I/O error. This is also the default mode in which file is opened.
Read and Write ('r+') : Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exists.
Write Only ('w') : Open the file for writing. For existing file, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exists.
Write and Read ('w+') : Open the file for reading and writing. For existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.
Append Only ('a') : Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
Append and Read ('a+') : Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

```
# A test program for writing to files
# the file we want to write to
a_filename = 'test.txt'
# Write a string to file
f = open(a_filename, 'w')
f.write("This is a test\n")
f.close()
# Read the string back
f = open(a_filename, 'r')
line = f.readline()
print(line)
f.close()
```

What do we read?

In Python, file reading **returns strings**

One line (→ returns '' if we are at the end) <some_string> = <file_object>.readline()
All lines as a list of strings (including line-endings like '\n') <some_list> = <file_object>.readlines()
The **entire file** as one string <some_string> = <file_object>.read()
At most <num_of_bytes> **number of bytes** <some_string> = <file_object>.read(<num_of_bytes>)

How can we read all the lines of a file?

```
# A test program for reading all lines of a file
# the file we want to read from
a_filename = 'test.txt'
# Read all lines in file and print to screen
f = open(a_filename, 'r')
for line in f:
        print(line, end='')
f.close() #important
```

FILES AND PATHS

Working with paths in Python

SW10-2.1-weg_zur_musik

- Use the os.path Python module
  import os
- Get the current directory
  os.getcwd()
- Compose path
  os.path.join(<my_directory>, <my_sub_directory>)
- Convert a relative path to an absolute path
  os.path.abspath(<a_relative_path>)
- List files in directory
  os.listdir(<a_path>)

```
path_to_file = os.path.abspath(os.path.join(os.getcwd(), '..', 'test.txt'))
```

HANDLING EXCEPTIONS

- If the file is not there, there will be an error in your program
- With the try – except construct, you can handle these exceptions and exit gracefully

Try and except

```
try:
	# Code that can fail (few code!)
except IOError [as <detail>]:
	# What we do if there is an IOError
	# The variable detail contains more information
except SomeOtherError [as <detail>]:
	# What we do if there is SomeOtherError
else:
	# Code if there is no Error
finally:
	# Always executed
```

```
# A file name, file does not exist
a_filename = "does_not_exist.txt"
# Use try to see if the file can be opened
try:
	f = open(a_filename, 'r')
except IOError:
	print(("Cannot open file: ") + a_filename)
else:
	with f: #with???
		print("** Reading the file: ")
		for line in f:
			print(line, end = '')
print("** done.")
```

- Open the file…
- …using a with block

- Use for to read all lines
- If you want to write, make sure you tell open() to enable writing

- The file is closed once you leave the with block

- Use the os module to work with paths, use os.path.join() to compose paths
- Handle exceptions with try-except

V11: Dictionaries

```
<my_dict> = {<a_key>: <a_value>, <another_key>: <some_value>, …}
```

One element of `my_dict` (= `<a_key>: <a_value>`)

➔ Dictionaries are iterable, too, but can be indexed by any kind of value

- **Access a value using a key** (but not vice versa!) (compare with a list: access a value using an index)
- Keys can have any type, but need to be immutable (strings, numbers, ...; not: lists)
- A dictionary **can't contain multiple elements with the same key** (but: values may be of type list…)
- **Values don't have to be unique**

lists, tuples and strings

What do lists, tuples (and strings) have in common?

➢ They are iterable objects that store a collection of single elements that can be addressed by a zero-based index.

Why do we have to be careful when copying lists?

- Because of shared references – when just binding a new name to an existing list, the new name refers to the same object in memory.

Why is this not a problem of tuples and strings?

- Because tuples and strings are immutable – thus, when changing the object, a new one is automatically (transparently) generated in the background

DICTIONARIES (ACCESSING VALUES USING A KEY)

```
# Phonetic alphabet definition
phonetic_alphabet = {
        'A': 'Alpha',
        'B': 'Bravo',
        'C': 'Charlie',
        # snip
        'Y': 'Yankee',
        'Z': 'Zulu'
        }
# Ask the user for a letter
letter = input('Enter a key [A-Z]: ')
letter = letter.upper()
if letter in phonetic_alphabet:
        print(letter + " is pronounced " + phonetic_alphabet[letter])
else:
        print("Sorry, " + letter + " not found")
```

Constructing a dictionary

Create a new dictionary: <a_dict> = {<key>: <value>, <next_key>: <next_value>, …} ➢ phonebook = {'jess': '079-777-8899', 'pete': '079-777-9988'}
Add a new entry: <a_dict>[<new_key>] = <new_value> ➢ phonebook['john'] = '079-777-1122'
Change an entry: <a_dict>[<key>] = <new_value> ➢ phonebook['jess'] = '079-777-1234'
Remove an entry: del <a_dict>[<key>] ➢ del phonebook['john']
Get and delete an entry in a single run: <value> = <a_dict>.pop(<key>) ➢ petes_number = phonebook.pop('pete')
Check if a key is present (i.e. a Boolean expression): <key> in <a_dict> ➢ 'jess' in phonebook
Get value of a key (default if not present): <value> = <a_dict>.get(<key>, <default>) ➢ petes_number = phonebook.get('pete', 'Not available')

Iterate through a dictionary:
<keys> = <a_dict>.keys() # also available: .values() or .items() → returns [(key,value),…]
for <key> in <keys>:
        print <a_dict>[<key>] # do whatever with this item
names = phonebook.keys()
for name in names:
        print(name + " : " + phonebook[name])

```
# Phonetic alphabet definition
phonetic_alphabet = {'A': 'Alpha', 'B': 'Bravo', ..., 'Y': 'Yankee', 'Z': 'Zulu'}
# Print the alphabet
print("Here is the phonetic alphabet")
letters = phonetic_alphabet.keys()
for letter in letters:
        print(letter + " : " + phonetic_alphabet[letter])
# Ask the user for his/her name
firstname = input("Write your firstname, I'll translate: ").upper()
for char in firstname:
        word = phonetic_alphabet.get(char, '?')
        print(word, '', sep=' – ', end='')
```

In fact, you have used object oriented features in Python:

```
phonebook = {'john': '079-777-1122', 'pete': '079-777-8899'}
phonebook.get('john', 'not known') # number if found, 'not known' otherwise
```

ON OBJECTS AND METHODS

Set

Sets can be used to eliminate duplicates in a list

```
>>> all_names = ['john', 'peter', 'jess', 'peter', 'tom', 'john']
>>> unique_names = set(all_names)
set(['john', 'peter', 'jess', 'tom'])
>>>
```

Example: Find restaurants that both peter and jess like

```
>>> peters_restuarants = ['bloom', 'mensa', 'kebab']
>>> jess_restaurants = ['migros', 'bloom', 'tibits']
>>> common = set(peters_restaurants) & set(jess_restaurants)
set(['bloom'])
>>>
```

JSON

JSON	Python
object { }	dict
array [ ]	list
string	unicode
number (int)	int, long
number (real)	float
true	True
false	False
null	None

Creating JSON – dump()

To file: json.dump(<object>, <file_identifier>)

To string: json.dumps(<object>)

```
import json
# define a dictionary
phonebook = {'jess': '079-777-8899', 'pete': '079-777-9988'}
print(json.dumps(phonebook, indent=4, separators=(';', ' --> ')))
```

```
{
"jess" --> "079-777-8899";
"pete" --> "079-777-9988"
}
```

Reading JSON – load()

```
import json
json_as_text = '''
{
        "Name": "McGregor",
        "Firstname": "John",
        "Points": [100, 90, 85]
}
'''
info = json.loads(json_as_text)
print(type(info))
print(info)
```

```
<class 'dict'>
{'Name': 'McGregor', 'Firstname': 'John', 'Points': [100, 90, 85]}
```

V12: Standard modules (1)

Dictionary, class and module

What is a dictionary?

- A kind of list where the index (called a key) can have any (immutable) type; a list of values assigned to keys

What is a class?

- In OO programming, a class is the source code (blueprint) of a new data type that contains the data itself as well as operations on it (called methods)

What is dict? (as in e.g. type(dict), help(dict) etc.)

- The class in Python that implements a dictionary

We have used modules many times already:

- import random
- import os
- import math

MODULE BASICS

To import a module:

- import <module_name> # module's filename without '.py' e.g., import math
- import <long_module_name> as <some_abbrev> e.g., import math as m
- If module is in another path:

```
import sys
sys.path.insert(0, <'/path/to/module'>)
import <module_name>
```

Working with modules

- Once loaded, access members of the module with a dot:
  - <module_name>.<function>()
  - e.g., math.exp()
- Individual functions or objects can be imported with:
  - from <module_name> import <function1> [, <function2>, …]
  - e.g., from math import exp()
- Individually imported functions can be used without module_name as prefix
- Show the content of a module:
  - dir(<module_name>)
  - e.g., dir(math)

- BTW: **dir() shows currently loaded/defined modules/variables**

SOME INTERESTING STANDARD MODULES, E.G. DATETIME & REGEX

Useful standard modules

os:	Operating system dependent functionality, e.g. os.path
sys:	System specific parameters and functions, e.g. sys.version_info
math:	Mathematical function, e.g. math.exp()
random:	Random number generator, e.g. random.randint()
time:	Timing and time related functionality, e.g. time.sleep()
datetime:	Date and time related functionality, e.g. datetime.date.today()
csv:	load comma separated files
json:	Javascript object notation encoder decoder, e.g. json.dump()
re:	Regular expression (see later)

Datetime

```
from datetime import date
today = date.today()
# Ask the user for his/her birthday
bday_day = int(input("What day is your birthday [1-31]? "))
bday_month = int(input("What month is your birthday [1-12]? "))
your_birthday = date(today.year, bday_month, bday_day)
# Check if it is next year
if your_birthday < today:
        your_birthday = your_birthday.replace(year = today.year + 1)
# Calculate the number of days until the bday
time_to_birthday = abs(your_birthday - today)
```

Import re

```
import re
str = 'phone: 079-777-1122'
# Usage: re.search(<regex_pattern>, <string_to_be_searched>)
match = re.search(r'\d{3}-\d{3}-\d{4}', str)
# If-statement after search() tests if it succeeded
if match:
        print('found', match.group()) # 'found a valid phone number'
else:
        print('did not find anything')
```

Selected characters with special meaning

. (a period) → matches any single character except newline ('\n')
\w → matches a "word" character: a letter or digit or underscore [a-zA-Z0-9_]
\s → matches a single whitespace character (space, newline, return, tab)
\d → matches a decimal digit [0-9]
[ ] → matches each of the set of characters contained in the brackets
{n} → modifies the previous a character: matches if it is repeated exactly n times
\ → inhibits the "specialness" of a character. E.g. \. Matches a regular dot
+ → modifier: 1 or more occurrences of pattern to its left, e.g. 'i+' = 1 or more "i"
* → modifier: 0 or more occurrences of the pattern to its left
? → modifier: match 0 or 1 occurrences of the pattern to its left
| → alternation (or): either the pattern to the left or to the right
() → used to group patterns in order to form sub-patterns (e.g., for |)
(these groups are returned by match.group() in the example on the previous slide)
^ → Beginning of string/line
$ → End of string/line

```
import re

filename = 'email-footer.txt'  # Load email footer from file
try:
    f = open(filename, 'r')
except IOError:
    print('Cannot open file: ' + filename)
    exit()
else:
    with f:
        text = f.read()
print(text + '\n')  # print the email footer
phone_match = re.search(r'((\+41\s)|(0))\d\d \d{3} \d{4}', text)  # regex string
if phone_match:  # see https://docs.python.org/2/library/re.html#match-objects
    print('found phone: ', phone_match.group())  # found a valid phone number
else:
    print('did not find phone number')
email_match = re.search(r'[\w.-]+@[\w.-]+', text)  # find my email address
if email_match:
    print('found email: ', email_match.group())  # found a valid email address
else:
    print('did not find email address')
```

YOUR OWN MODULE

- You can create your own module (named e.g. my_module) by
  - writing the relevant functions and attributes/variables
  - in a file called my_module.py
- Make sure to include
  - docstring documentation of your functions
  - a variable __version__='0.1' containing the version number of your module
- Load the module with import my_module as my (from a script in the same directory as my_mdoule)
- Access the functions with e.g. my.say_hello()
- Try dir(my) and help(my) to explore the module

```
phonetic_alphabet = {'A': 'Alpha', …, 'Z': 'Zulu'}
def string_to_phonetic(string):
        ''' Looks up the characters of string in the ... '''
        phonetic = ""
        for char in string:
                word = phonetic_alphabet.get(char.upper(), '?')
                phonetic += word + ' –'
        return phonetic
def say_hello(name):
        ''' prints hello 'name' '''
        print("Hello " + name)
def print_dictionary(dictionary):
        ''' prints keys/values to screen '''
        keys = dictionary.keys()
        for key in keys:
                print(key, ":", dictionary[key])
''' The version number '''
__version__ = "0.1
```

```
>>> import v12_my_module as my
>>> dir(my)
['__builtins__', '__doc__',
'__file__', '__name__',
'__package__', '__version__',
'phonetic_alphabet',
'print_dictionary', 'say_hello',
'string_to_phonetic']
>>> help(my.say_hello)
Help on function say_hello in
module v12_my_module:
say_hello(name)
prints hello 'name'
>>> my.say_hello('Thilo')
Hello Thilo
>>>
my.string_to_phonetic('Thilo')
Tango –Hotel –India –Lima –Oscar
```

V13: Standard and external modules (2)

EXTERNAL MODULES AND PACKAGE MANAGERS

About external Python packages/modules

In addition to standard modules included in Python, there are many external packages. They need to be downloaded, possibly compiled and installed:

Python needs to know where to look for packages

• Generally, Python searches the $PYTHONPATH environment variable when you type import module_name

• The current working directory (cwd) is also searched

READ, MANIPULATE AND PLOT DATA: INTRODUCING NUMPY

General Data Analysis pipeline

1. Read data (csv, xlrd, json, pandas, etc.)
2. Manipulate:
   - Convert, arrange, wrangle data (numpy, pandas, etc.)
   - Analyze data, produce results (numpy, scipy, sklearn)
3. Output results (print, csv, xlrd, pandas, etc.)
4. Visualize data and results (matplotlib, chaco, bokeh, vtk, etc.)

Background on NumPy

- NumPy provides convenient array manipulation routines (like e.g. Matlab®)
  - Possesses useful linear algebra (vector and matrix operations), Fourier transform, andrandom number capabilities
  - NumPy is highly optimized and provides fast execution (→ see Appendix)
- Transitioning from Matlab® to NumPy is straightforward

NumPy

Load the module: import numpy as np

• x = np.array([1,2,3,4,5])	# Create a 1D array based on a list	[1 2 3 4 5]
• x = np.zeros(10)	# Create a 1-D array of 10 zeros	[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
• x = np.zeros(10).reshape(5,2)	# Create a 5 row times 2 column matrix of zeros	[[0. 0.] [0. 0.] [0. 0.] [0. 0.] [0. 0.]]
• x = np.ones([5,2])	# Create a 5 row times 2 column matrix of 1s	[[1. 1.] [1. 1.] [1. 1.] [1. 1.] [1. 1.]]
• x = np.linspace(0,1,10)	# Create a 1-D array from 0 to 1included with 10 equidistant values	[0. 0.11111111 0.22222222 0.33333333 0.44444444 0.55555556 0.66666667 0.77777778 0.88888889 1.]

More NumPy

```
[ , , ] Selection
[ : : ] Slicing
a[0,3:5] ► [ 0th col, 3:5-1 ]
```

Getting dimensions of arrays

```
x.ndim  # Number of dimensions
x.shape # Number of elements
          in each dimension
```

Accessing values with slicing
(same as with Python lists)

```
x[1:] # leave out the first element
x[:2] # get the first two
```

```
>>> a[0,3:5]
array([3,4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])

>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

Simple operations (+ - * /) operate **element-wise** on array values (unlike Matlab®)

```
z = x + y # add corresponding elements in x and y, save in z
z = x * 5 # multiply each element by 5
```

Concatenate two arrays

```
z = np.hstack([x,y]) # horizontally
z = np.vstack([x,y]) # vertically
```

x y

x
y

INTRODUCING MATPLOTLIB

Background on Matplotlib

Matplotlib provides 2D (and some 3D) plotting functionality

- Includes: x-y plots, histogram, bar and pie plots – and many more
- Syntax is similar to Matlab®

<u>General workflow:</u>

1. Import Matplotlib: import matplotlib.pyplot as plt
2. Get data (e.g. with NumPy)
3. Generate new figure: plt.figure()
    1. Plot data: plt.plot()
    2. Set title, x and y axis label: plt.title(), plt.xlabel(), plt.ylabel()
    3. (optional) Other formating, e.g.: plot.grid(True)
4. Make plot visible: plt.show()

```
# 1. Import
import matplotlib.pyplot as plt
import numpy as np
# 2. Get the data
x = np.linspace(0, 360, 90) # 0 .. 360 every 4 degrees
y = np.sin(x * np.pi / 180) # convert deg to radian
# 3. Create Figure
plt.figure()
# 3.1 Plot data
plt.plot(x,y)
# 3.2. Set title, x and y label
plt.title('A sine wave')
plt.xlabel('Angle [deg]')
plt.ylabel('sin(x)')
# 3.3 Other formating
plt.grid(True)
# 4. Make plot visible
plt.show()
```

INTRODUCING XLRD

Xlrd (Excel read) in a nutshell

1. Open the file: wb = xlrd.open_workbook()
2. Select a sheet: sheet = wb.sheet_by_index(0)
3. (optional) Check dimensions: (sheet.nrows, sheet.ncols)
4. Read data: iterate over rows and access values in cells

```
for i in range(sheet.nrows):
    sheet.cell(i, 0).value # ith row, 0th column
```

Good to know

- Each cell has a type and a value
- Access entire rows with sheet.row(<row>)and columns with sheet.col(<col>)

Ursula Mayer

PUTTING IT ALL TOGETHER: KINEMATICS OF A TOY CAR

"A spring loaded toy car is filmed with a high speed camera at 240 frames per second while accelerating."
What is the average velocity and acceleration of the car?
Available information:

- Video contains a visual reference distance
- Position of the car was measured with a ruler on screen at every 24 frames and saved to an excel file
- The 50cm real distance were 20cm on (a certain) screen

1. Load data from excel sheet (xlrd)
2. Convert data:
    1. on screen progress → real distance
    2. frame number → seconds
3. Compute velocity and acceleration from position (numpy)
    1. velocity = change in position per time (derivative, approx. by finitedifference)
    2. acceleration = change in velocity per time (derivative, approx. by finitedifference)
4. Calculate average velocity and acceleration (numpy)
5. Plot the results as x-y plots (matplotlib)

```
# Reference: http://serc.carleton.edu/dmvideos/activities/example10.html
import numpy as np
import matplotlib.pyplot as plt
import xlrd

# Conversion constants
screen_to_real_ratio = 20.0 / 50.0  # 20cm on screen are 50cm in real life
                                      (depends on screen size!)
frames_per_second = 240  # used to convert frame number to seconds


''' 1: Load data from excel sheet (xlrd) '''
# Read frame number and screen distance from Excel sheet
wb = xlrd.open_workbook('v13_toycar.xls')
sheet = wb.sheet_by_index(0)

frames = np.zeros(sheet.nrows-1)  # notice the -1, we leave out the header
screen_distance_mm = np.zeros(sheet.nrows-1)
for i in range(1, sheet.nrows):
    frames[i-1] = float(sheet.cell(i, 0).value)
    screen_distance_mm[i-1] = float(sheet.cell(i, 1).value)
```

```
''' 2: Convert data '''
# Convert frame number to seconds
# (note the vectorized computation here and below: the result is a NumPy array!)
times = frames / frames_per_second
# Convert screen distance to actual distance in meters
actual_distance_m = screen_distance_mm / (screen_to_real_ratio * 1000)

''' 3: Compute velocity and acceleration from position (numpy) '''
# Calculate velocity and acceleration from position using finite differences.
delta_t = times[1] - times[0]
velocity_m_per_s = np.diff(actual_distance_m) / delta_t
acceleration_m_per_s2 = np.diff(actual_distance_m, 2) / (delta_t * delta_t)

''' 4: Calculate average velocity and acceleration (numpy) '''
# Calculate average velocity and acceleration
mean_velocity = np.mean(velocity_m_per_s)
mean_acceleration = np.mean(acceleration_m_per_s2)
```

```
''' 5: Plot the results as x-y plots (matplotlib) '''
# Distance vs Time
plt.figure()
plt.plot(times, actual_distance_m, marker='*')
plt.title('Distance vs time')
plt.xlabel('time [s]')
plt.ylabel('distance [m]')
plt.grid(True)

# Velocity vs Time
plt.figure()
plt.plot(times[1:], velocity_m_per_s, marker='*')
plt.title('Velocity vs time, mean = ' + str(round(mean_velocity, 2)) + ' m/s')
plt.xlabel('time [s]')
plt.ylabel('velocity [m/s]')
plt.ylim(ymin=0)
plt.grid(True)
```

```
# Acceleration vs Time
plt.figure()

plt.plot(times[2:], acceleration_m_per_s2, marker='*')
plt.title('Acceleration vs time, mean = ' +
          str(round(mean_acceleration, 2)) + ' m/s^2')

plt.xlabel('time [s]')
plt.ylabel('acceleration [m/s^2]')
plt.ylim(ymin=0)
plt.grid(True)

plt.show()
```

Pure Python vs. NumPy execution time

```
import time
def traditional_version():
    t1 = time.time()
    X = range(10000000)
    Y = range(10000000)
    Z = []
    for i in range(len(X)):
        Z.append(X[i] + Y[i])
    return time.time() - t1
```

```
import time
import numpy
def numpy_version():
    t1 = time.time()
    X = numpy.arange(10000000)
    Y = numpy.arange(10000000)
    Z = X + Y
    return time.time() - t1
```

NumPy array methods

Get mean, standard deviation, sum, min, max of an array

- x.mean() # or np.mean(x), x.std() or np.std(x)
- x.sum() # or np.sum(x)
- x.min() # or np.min(x), x.max() or np.max(x)

Get mean in 2-D array for all columns

- x.mean(axis=0) # or np.mean(x, axis = 0)

Get mean in 2-D array for all rows

- x.mean(axis=1) # or np.mean(x, axis = 1)

Index of min or max

- x.argmin() # or np.argmin(x), x.argmax() or np.argmax(x)

Sorting an array

- x.sort() # or np.sort(x)

Deep copy an array

- x.copy() # or np.copy(x)

FAQ

V14: Python Patterns

TYPICAL PATTERNS IN PYTHON

"Accumulator"

Applicable to: Accumulators of type int/float/string/list/dict

```
accumulator_variable = empty
for element in some_list:
accumulator_variable += element.value
do_something(accumulator_variable)
```

```
pax_list = [19, 23, 36, 31]
sum = 0
for pax_age in pax_list:
sum += pax_age
average = sum/float(len(pax_list))
print(average)
```

"Infinite loop"

```
while True:
    if some_complex_expression == True:
        break
```

```
while True:
    str_result = input('an integer, please: ')
    if str_result.is_digit():
        break
```

"For-Each style loop vs. index-based loop"

Apply for-each if: You need access to all the elements of the iterable, one element at a time	Apply index-based loop if: You need access to the index (i.e., the place of storage) or to more than one element during the same loop iteration

some_list = [...]
for element in some_list:
    do_something(element))
	some_list = [...]
	for i in range(len(list_length)):
	    do_something(some_list[i])

"Refactor code to a function"

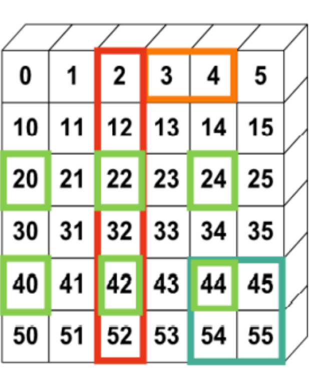
Key ingredients:

- Choose a meaningful function name; maybe change names of parameters
- Return a result
- "Catch" this result when calling the function in the place of the original spaghetti-code

"Traverse a 2D list"

```
for row in matrix:
    for cell in row:
        do_something(cell)
# or, preferably:
for y in range(len(matrix)):
    for x in range(len(matrix[y])):
        do_something(matrix[y][x])
```

```
>>> a[0,3:5]
array([3,4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])

>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```



"String operations and slicing"

Best practices:

- Use slicing syntax to create substrings
  E.g. leave the last n characters out with my_string[:-n]
- Remember basic string methods to ease different tasks
  E.g. upper(), lower(), is_digit(), split(), replace()
- Build strings by concatenating two strings with + (maybe convert operands to string first)
  E.g. new_string = "The answer is " + str(42) + '.'
- First build strings, then do further processing
  (e.g., returning them from a function; printing them to the console; writing them to a file)

Data type change after applying an operator

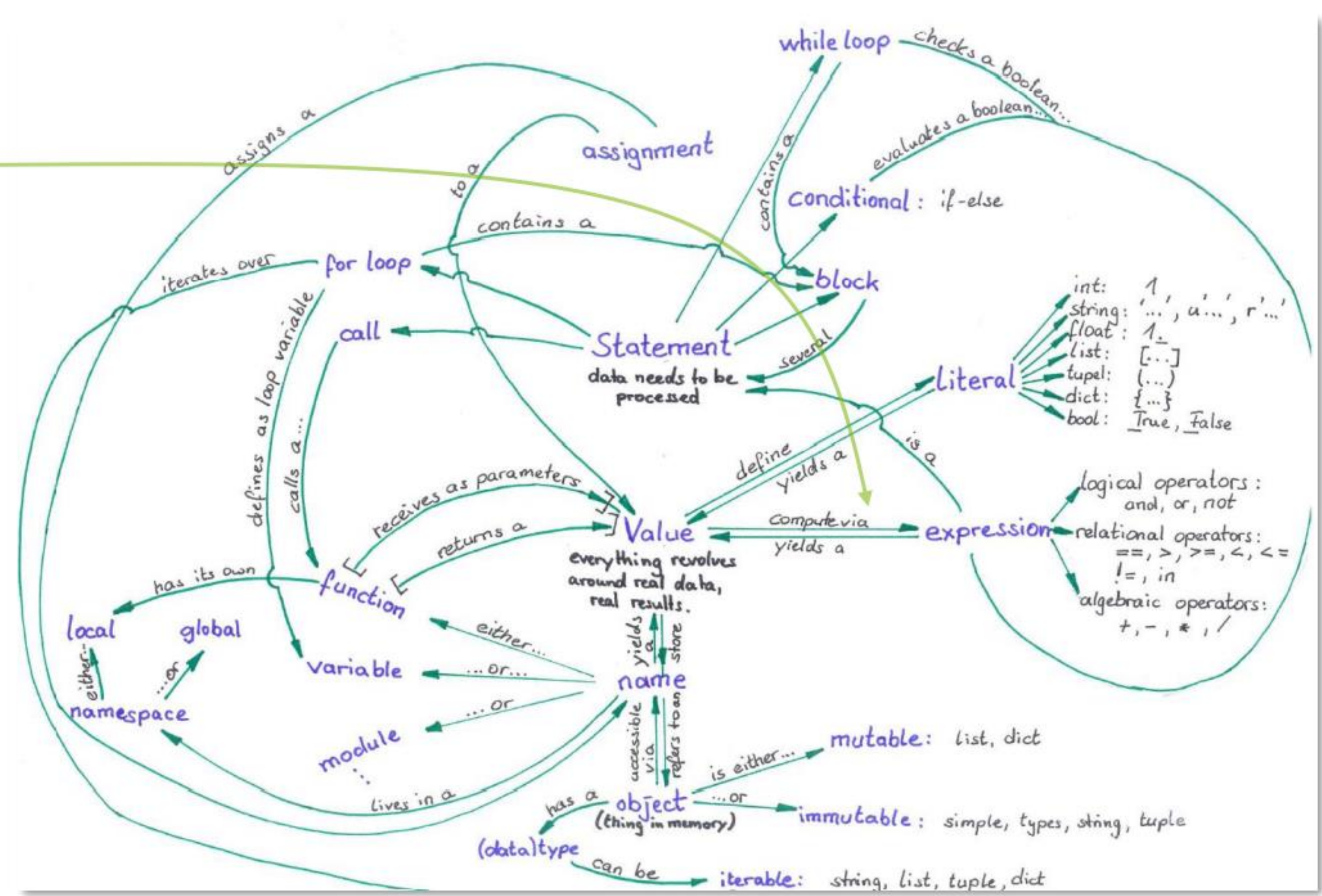
- Comparison (==) yields a bool
- Multiplication of int and string yields a string (string gets replicated int times)

RECAP: INTERRELATION OF PYTHON CONCEPTS

Let's create a concept map:

For example, values are computed using expressions, those yield values when evaluated

➔ An expression can stand anywhere a value is expected (or vice versa)!



General strategy for successful scripts

1. Think about data structures
   - Can the problem be stored in some kind of iterable (string, list etc.)?
2. Think about processing these data structures
   - Iterables can efficiently and conveniently be mass-processed using for-loops

➔ Choose a data structure for which you can think of an efficient & convenient way of processing!

➔ Often, this resorts to lists and loops.