EINFÜHRUNG IN DIE PROGRAMMIERUNG

VARIABLES, CALCULATIONS, STRINGS

DHBW MANNHEIM
WIRTSCHAFTSINFORMATIK (DATA SCIENCE)

Markus Menth
Martin Gropp

VARIABLES

VARIABLES

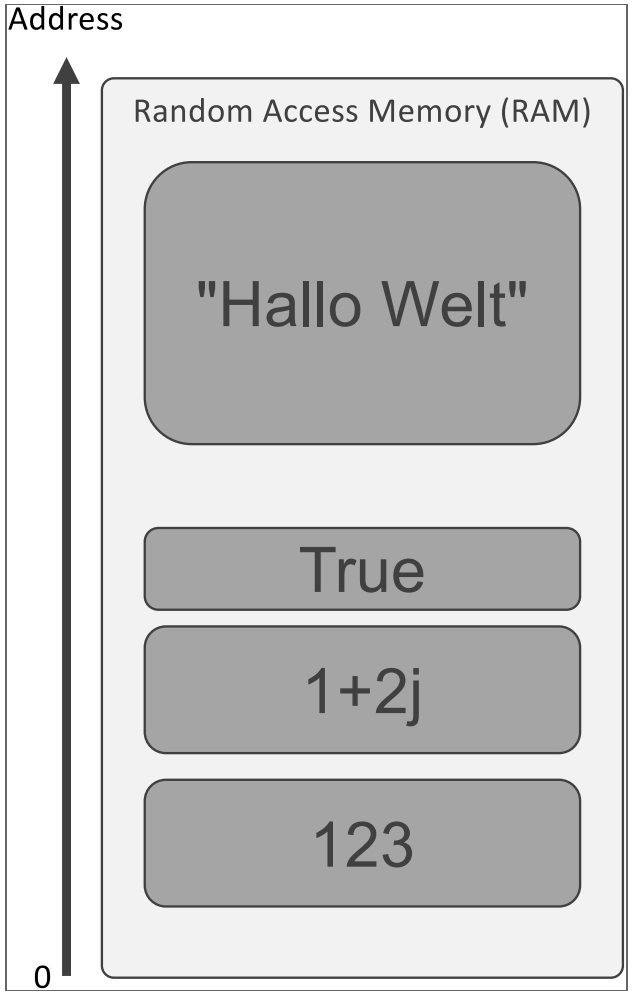
Variables are storage locations for data
These locations are identified using a name

- They have a certain (fixed or dynamic) size and type
- They are stored in RAM of a process

Variables can be accessed (read) and modified at any time

- They form the basis for the calculations of programs

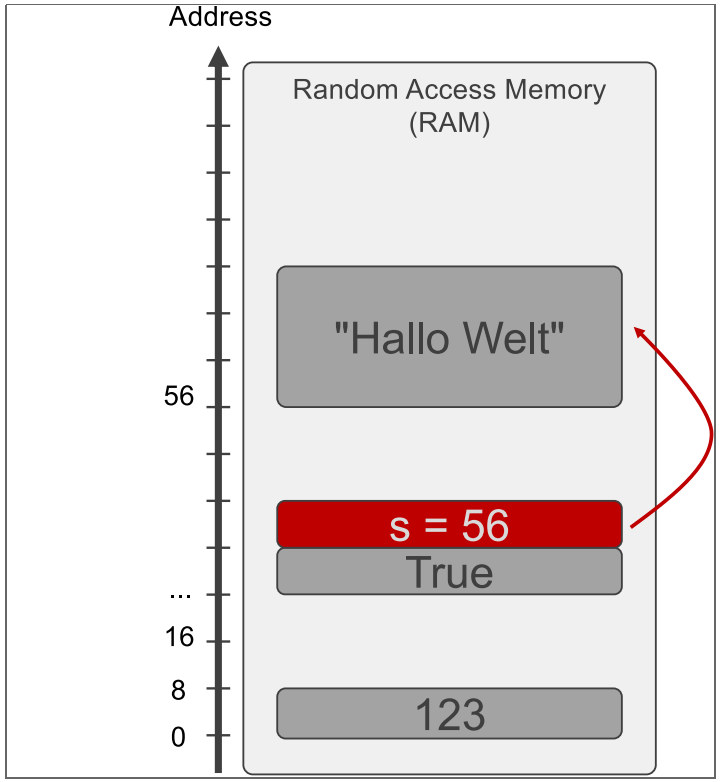
VARIABLES AS STORAGE LOCATIONS



VARIABLES AS REFERENCES TO OBJECTS 

In many programming languages variables can be references

- They only reference another object in memory
- → separate chapter



DECLARATION OF VARIABLES

Variables have unique identifiers

- Names of variables are case-sensitive
- There exist **rules regarding names of variables**
- Identifiers must not be reserved words

Examples

```
x = 1
y = 2
s = 'Hallo Welt'
x = 2
```

PYTHON: RESERVED WORDS

Python has a set of reserved words:

- They are part of the language
- They are not the same as built-in functions

RESERVED WORDS

and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	
class	finally	is	return	
continue	for	lambda	try	
def	from	nonlocal	while	

NAMING CONVENTION FOR VARIABLES

Basic rules:

- Variable names start with a letter, followed by letters, numbers, or underscores.
- Use a single letter, a word or multiple words, separated by underscores.
- Use only lowercase letters.

```
x = 1                 # single letter
word = 2              # word
word2 = 3
multiple_words = 4    # words
```

Variable names should be descriptive.
Only use single-letter names for short-running, "local" variables.

TYPES OF CONTENTS OF VARIABLES

Most programming languages have different types of variable contents.
Python's principal **built-in types** are:

- numerics: `int`, `float`, `complex`
- truth values: `bool`
- text: `str`
- sequences: `list`, `tuple`, `range`
- `set`
- mappings: `dict`
- classes, instances, exceptions, modules, functions, …

STATIC VS. DYNAMIC TYPING 🔬

Some languages require variables to have a fixed type: **static typing**
Python uses **dynamic typing**:
Typing decisions happen at run-time (instead of at compile-time)
Advantages? Disadvantages?

STATIC VS. DYNAMIC TYPING 🔬

Dynamic typing:

- + flexibility
- + no need to specify the type of a variable: its type can change at runtime
- – errors may come to light only under certain circumstances at runtime
- – less efficient

DUCK TYPING 🔬

"duck-typing"

> *"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."*
> *(James Whitcomb Riley)*

Most languages look at an object's type to determine if an object can be used in some context.
Python only requires that an object have all used *methods* and *attributes*.
(More about this later.)

LITERALS

LITERALS

You can think of a literal as a constant value of a certain type.

> *Literals are notations for constant values of some built-in types. (**Python Docs - Literals**)*

= values that are part of the program code.

INTEGER LITERALS

EXAMPLES

```
10
-12345
1_234_567
```

DIFFERENT NUMBER SYSTEMS 🔬

```
0xFF    # hexadecimal (base 16): 255
0o70    # octal (base 8): 56
0b1011  # binary (base 2): 11
```

FLOAT LITERALS

- Based on the IEEE 754 floating point standard
- Two different types of notation: standard and **"scientific"**

Scientific notation:

- `1.5e2` means $1.5 \cdot 10^{2}$
- `.5e-2` means $0.5 \cdot 10^{-2}$

EXAMPLES

```
10.5
.5
-1.
1_234.567

1.5e2
.5e-4
```

COMPLEX LITERALS

Comprised of real and imaginary part (of type `float`).

```
1+2j
3.14j
0-3j
```

(Complex numbers are not supported everywhere!)

HOW TO CALCULATE IN PYTHON

Operator	Meaning	Example
+	Add two operands or unary plus	x + y *or* +2
-	Subtract right operand from the left or unary minus	x – y *or* –2
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y

HOW TO CALCULATE IN PYTHON

Operator	Meaning	Example
%	Modulus (remainder of the division of left operand by the right)	x % y
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x ** y

EXAMPLE

```
x = 2 ** 2
y = 16 // 6
z = 3 * (y % 3)
```

EXERCISE

What are the last and second last digits of a number?

SOLUTION

What are the last and second last digits of an (integer) number?

```
>>> n = 12345
>>> n % 10
5

>>> n2 = n // 10
>>> n2 % 10
4
```

HOW TO CALCULATE IN PYTHON

Python's `math` module has more useful functions and constants.
It needs to be imported before use:

```
import math
```

- `math.sqrt(x)`
- `math.sin(x)`
- `math.exp(x)`
- `math.log(x)`
- …
- `math.pi`
- `math.e`

EXERCISE

- Store the radius r of a circle in a variable `r`
- Compute the area of the circle ($\pi \cdot r^2$)
- Store the result in the variable `area`
- Print the result on the console

EXERCISE

Write a Python program to compute the distance between the points $(x1, y1)$ and $(x2, y2)$

- Use the **Pythagorean theorem**
- Use **math.sqrt** from the math module:
  add `import math` to the beginning of your python file.

SOLUTION

```
import math

x1 = 0
y1 = 0
x2 = 1
y2 = 1

print(
    math.sqrt(
        ((y2-y1) ** 2) +
        ((x2-x1) ** 2)
    )
)
```

BOOLEAN LITERALS / TRUTH VALUES

- Used for distinct true and false values
- Used for conditions (later)

```
x = True
y = False
```

STRINGS

STRING LITERALS

Enclosed in matching single (`'`, preferred) or double quotes (`"`):

```
'Hallo Welt!'
"Hallo Welt!"
```

- But not `"Hallo Welt'`

String literals are of type **str**.

MULTI-LINE STRINGS

Alternative representation for longer strings:
text enclosed in matching groups of triple quotes (""" or ''')

```
multi = """Hallo Welt.
 Es ist total schön hier"""
```

PRINTING STRING VARIABLES

Careful! Strings are written with quotation marks, string variables aren't!

```
s = 'foo'
print(s)
print('s')
```

```
foo
s
```

STRING ENCODING

On a computer, characters are (of course) represented as numbers.
There are many (of course) incompatible standards.
Common: ASCII, UTF-8

STRING ENCODING: ASCII

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	*[NULL]*
1	1	1	1	*[START OF HEADING]*
2	2	10	2	*[START OF TEXT]*
3	3	11	3	*[END OF TEXT]*
4	4	100	4	*[END OF TRANSMISSION]*
5	5	101	5	*[ENQUIRY]*
6	6	110	6	*[ACKNOWLEDGE]*
7	7	111	7	*[BELL]*
8	8	1000	10	*[BACKSPACE]*
9	9	1001	11	*[HORIZONTAL TAB]*
10	A	1010	12	*[LINE FEED]*
11	B	1011	13	*[VERTICAL TAB]*
12	C	1100	14	*[FORM FEED]*
13	D	1101	15	*[CARRIAGE RETURN]*
14	E	1110	16	*[SHIFT OUT]*
15	F	1111	17	*[SHIFT IN]*
16	10	10000	20	*[DATA LINK ESCAPE]*
17	11	10001	21	*[DEVICE CONTROL 1]*
18	12	10010	22	*[DEVICE CONTROL 2]*
19	13	10011	23	*[DEVICE CONTROL 3]*
20	14	10100	24	*[DEVICE CONTROL 4]*
21	15	10101	25	*[NEGATIVE ACKNOWLEDGE]*
22	16	10110	26	*[SYNCHRONOUS IDLE]*
23	17	10111	27	*[ENG OF TRANS. BLOCK]*
24	18	11000	30	*[CANCEL]*
25	19	11001	31	*[END OF MEDIUM]*
26	1A	11010	32	*[SUBSTITUTE]*
27	1B	11011	33	*[ESCAPE]*
28	1C	11100	34	*[FILE SEPARATOR]*
29	1D	11101	35	*[GROUP SEPARATOR]*
30	1E	11110	36	*[RECORD SEPARATOR]*
31	1F	11111	37	*[UNIT SEPARATOR]*
32	20	100000	40	*[SPACE]*
33	21	100001	41	**!**
34	22	100010	42	**"**
35	23	100011	43	**#**
36	24	100100	44	**$**
37	25	100101	45	**%**
38	26	100110	46	**&**
39	27	100111	47	**'**
40	28	101000	50	**(**
41	29	101001	51	**)**
42	2A	101010	52	*****
43	2B	101011	53	**+**
44	2C	101100	54	**,**
45	2D	101101	55	**-**
46	2E	101110	56	**.**
47	2F	101111	57	**/**

Decimal	Hexadecimal	Binary	Octal	Char
48	30	110000	60	**0**
49	31	110001	61	**1**
50	32	110010	62	**2**
51	33	110011	63	**3**
52	34	110100	64	**4**
53	35	110101	65	**5**
54	36	110110	66	**6**
55	37	110111	67	**7**
56	38	111000	70	**8**
57	39	111001	71	**9**
58	3A	111010	72	**:**
59	3B	111011	73	**;**
60	3C	111100	74	**<**
61	3D	111101	75	**=**
62	3E	111110	76	**>**
63	3F	111111	77	**?**
64	40	1000000	100	**@**
65	41	1000001	101	**A**
66	42	1000010	102	**B**
67	43	1000011	103	**C**
68	44	1000100	104	**D**
69	45	1000101	105	**E**
70	46	1000110	106	**F**
71	47	1000111	107	**G**
72	48	1001000	110	**H**
73	49	1001001	111	**I**
74	4A	1001010	112	**J**
75	4B	1001011	113	**K**
76	4C	1001100	114	**L**
77	4D	1001101	115	**M**
78	4E	1001110	116	**N**
79	4F	1001111	117	**O**
80	50	1010000	120	**P**
81	51	1010001	121	**Q**
82	52	1010010	122	**R**
83	53	1010011	123	**S**
84	54	1010100	124	**T**
85	55	1010101	125	**U**
86	56	1010110	126	**V**
87	57	1010111	127	**W**
88	58	1011000	130	**X**
89	59	1011001	131	**Y**
90	5A	1011010	132	**Z**
91	5B	1011011	133	**[**
92	5C	1011100	134	****
93	5D	1011101	135	**]**
94	5E	1011110	136	**^**
95	5F	1011111	137	**_**

Decimal	Hexadecimal	Binary	Octal	Char
96	60	1100000	140	**`**
97	61	1100001	141	**a**
98	62	1100010	142	**b**
99	63	1100011	143	**c**
100	64	1100100	144	**d**
101	65	1100101	145	**e**
102	66	1100110	146	**f**
103	67	1100111	147	**g**
104	68	1101000	150	**h**
105	69	1101001	151	**i**
106	6A	1101010	152	**j**
107	6B	1101011	153	**k**
108	6C	1101100	154	**l**
109	6D	1101101	155	**m**
110	6E	1101110	156	**n**
111	6F	1101111	157	**o**
112	70	1110000	160	**p**
113	71	1110001	161	**q**
114	72	1110010	162	**r**
115	73	1110011	163	**s**
116	74	1110100	164	**t**
117	75	1110101	165	**u**
118	76	1110110	166	**v**
119	77	1110111	167	**w**
120	78	1111000	170	**x**
121	79	1111001	171	**y**
122	7A	1111010	172	**z**
123	7B	1111011	173	**{**
124	7C	1111100	174	**|**
125	7D	1111101	175	**}**
126	7E	1111110	176	**~**
127	7F	1111111	177	*[DEL]*

STRING ENCODING

ASCII (American Standard Code for Information Interchange) contains only 128 characters (7 bits), and some are special non-printable control characters.
→ no encoding for äöüß!
Consequence: many incompatible extensions
ISO 8859-1 ... ISO 8859-16, Windows-1250 ... Windows-1258, ...

UNICODE & UTF-8

The Unicode standard attempts to define a single standard with more than 1 million characters.
UTF-8 is a Unicode encoding that is backwards compatible with ASCII and is the standard encoding in Python.

- 0..127: ASCII characters, one byte per character
- >127: multiple bytes per character

a	97	61
ä	195 164	C3 A4
𝄞	240 157 132 158	F0 9D 84 9E

ESCAPING

Some characters in strings have a special meaning

- E.g. `"` or `'` terminate the string
- Question: how to represent `"` in a string?

Backslash `\` has a special meaning

- Escapes characters that otherwise have a special meaning
- Examples: newline, backslash itself, or the quote characters

ESCAPE SEQUENCES

Escape Sequence	Meaning
`\\`	**Backslash (\)**
`\'` and `\"`	**Single (') and double quote (")**
`\b`	ASCII Backspace (BS)
`\n`	**ASCII Linefeed (LF)**
`\r`	ASCII Carriage Return (CR)
`\t`	ASCII Horizontal Tab (TAB)
`\ooo`	Character with octal value `ooo`
`\xhh`	**Character with hex value** hh
`\N{name}`	Character named name in the Unicode database
`\uxxxx`	**Character with 16-bit hex value** `xxxx`

Escape Sequence	Meaning
\Uxxxxxxxx	Character with 32-bit hex value xxxxxxxx

Source: **Python 3 Documentation**

EXAMPLES OF ESCAPE SEQUENCES

Examples

```
print("\a")
print("a\bc")
print("Das ist ein Backslash: \\")
print("Das ist ein doppeltes Anführungszeichen: \"")
print("Hallo\tWelt\n.Hier ist es aber schön")
print("\123")
print("\xAB")
```

Unicode examples from the **Full Emoji List**

```
print("\U0001F62C")
print("\U0001F4A9")
print("\U0001F44B")
print("\U0001F98E")
```

STRING-OPERATIONS

String concatenation

- Strings are immutable
- Strings can be combined (to new objects) using **+**:
  `"Hello " + "World"`
- Python does **not** (unlike other languages) automatically convert other types to strings:
  `"Hello" + 1` gives an error
  (→ Type Conversion)

ACCESSING STRING CHARACTERS

- The individual characters of a string can be accessed using the `[]` operator: `s[index]`
- ⚠ Zero-based indexing: `s[0]`, `s[1]`, ...

EXAMPLE

```
s = 'hallo'
```

h	**a**	**l**	**l**	**o**
s[0]	s[1]	s[2]	s[3]	s[4]

ACCESSING STRING CHARACTERS

If the index is out of bounds for the string, Python raises an error:

```
>>> s = 'hallo'
>>> s[9]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

ACCESSING STRING CHARACTERS

Negative values start counting from the end of the string.

h	**a**	**l**	**l**	**o**
s[-5]	s[-4]	s[-3]	s[-2]	s[-1]

STRING SLICES

Handy way to extract parts of a string

- Syntax: `s[start:end]`
- Slice starts at position `start` (inclusive) and ends at `end` (exclusive)
- Omitting `start` or `end` defaults to start or end of the string

STRING SLICES

EXAMPLES

h	a	l	l	o
s[0]	s[1]	s[2]	s[3]	s[4]

```
s = 'hallo'
s[1:4]    # 'all'
s[1:]     # 'allo'
s[:]      # 'hallo' # -> this is a copy of the original string
s[1:100]  # 'allo'  # -> 100 is truncated to the length of the
string
```

STRING SLICES

String slices may also use negative numbers

- Just like negative indices

h	a	l	l	o
s[0]	s[1]	s[2]	s[3]	s[4]
s[-5]	s[-4]	s[-3]	s[-2]	s[-1]

```
s = 'hallo'
s[:-3]    # 'ha'
s[-3:]    # 'llo'
```

STRING OPERATIONS

Length of a string

- **len** returns the length of a string:

```
>>> len("Hello")
5
```

Others (**see documentation**)

- `s.lower()`, `s.upper()`, `s.strip()`, `s.isalpha()`, `s.isdigit()`, `s.isspace()`, `s.startswith('other')`, `s.endswith('other')`, `s.find('other')`, `s.replace('old', 'new')`, `s.split('delim')`, `s.join(list)`

Additional reading: **Python Strings (Google Developers)**

STRING FORMATTING

In order to insert specific values in a string there are to main options:

- f strings
- *format* method

Both options use curly braces to designate placeholders: {}

EXAMPLES

```
first_name = 'John'
last_name = 'Doe'

# f string
print(f'Hello, {first_name}!')

# format method
print('Hello, {}!'.format(first_name))
print('Hello, {} {}!'.format(first_name, last_name))
print('Hello, {1}, {0}!'.format(first_name, last_name))
print('Hello, {name}!'.format(name=first_name))
```

STRING FORMATTING 🔬

Advanced usage: see **Format Strings**

EXAMPLES

```
>>> import math
>>> f'pi is close to {math.pi:.2g}'
'pi is close to 3.1'

>>> n = 42
>>> f'The answer is {n:08d}'
'The answer is 00000042'
```

BYTES LITERALS 🔬

Python's **bytes** type represents "raw" uninterpreted data.
Bytes literals are similar to strings

- Are prefixed with `b`
- Example: `b"Hallo Welt"`

Properties

- Only ASCII characters are permitted (0-127)
- Other bytes are represented using escape sequences

```
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
64512

>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x04\x00'
```

TYPE CONVERSIONS

TYPE

With the *type()* method you can read the type of a variable.

```
x = "Hello World"
y = 1
z = 1.0

print(type(x))      # <class 'str'>
print(type(y))      # <class 'int'>
print(type(z))      # <class 'float'>
```

TYPE CONVERSION

Type Conversion ("casting") is a way to change the type of a variable.

INT

```
x = int(4)      # x will be 4
y = int(3.86)   # y will be 3
z = int("16")   # z will be 16
```

FLOAT

```
x = float(1)      # x will be 1.0
y = float("13.3") # w will be 13.3
```

STR

```
x = str("s1") # x will be 's1'
```

```
y = str(5)    # y will be '5'
z = str(3.7)  # z will be '3.7'
```

TYPE CONVERSION

EXAMPLE

`input(...)` will return the user input as a `str`.
If you need the user to enter a number, you need to convert it:

```
s = input('Please enter a number: ')
print(type(s))

n = int(s)
print(type(n))
print(n + 1)
```

EXERCISES

EXERCISE

Write a Python program to print the following string to the console:

```
Sample string:
a string that you "don't" have to escape
This
is a ....... multi-line
string --------> example
```

Hints

- Use a single statement
- Hint: maybe a multi-line string helps

SOLUTION

```
print("""Sample string :
a string that you "don't" have to escape
This
is a ....... multi-line
string --------> example
""")
```

EXERCISE

Swap the contents of two variables x and y

SOLUTION

```
x = 1
y = 2

tmp = x
x = y
y = tmp
```

There is also a more "pythonic" solution (details on why this works later!):

```
x, y = y, x
```

EXERCISE

Write a Python program with two `print` statements

- The first prints `Hallo`, the second `Welt`
- Don't print a newline after the first one
- Read the documentation of **print**

SOLUTION

```
print("Hallo", end=' ')
print("Welt")
```

AUFGABE: WIND-CHILL-FORMEL

Niedrige Temperaturen fühlen sich bei Wind oft noch kälter an als sie ohnehin schon sind. Ausgedrückt wird dies in der empirischen Wind-Chill-Formel. Die Formel gilt für kalte Temperaturen und ergibt eine Vergleichstemperatur, die bei schwachem Wind ähnlich kalt empfunden würde:

$$\mathrm{WCT} = A + B \cdot T + C \cdot v^{0,16} + D \cdot T \cdot v^{0,16}$$

T ist dabei die tatsächliche Temperatur (in °C), v die Windgeschwindigkeit (in km/h). Die Werte der Konstanten sind:

$$A = 13{,}12, B = 0{,}6215, C = -11{,}37, D = 0{,}3965.$$

Legen Sie Variablen für A, B, C und D an. Fragen Sie den Benutzer nach den Werten von T und v. Konvertieren Sie die Benutzereingaben in einen sinvollen Typ und berechnen Sie WCT!

LÖSUNG

$$\mathrm{WCT} = A + B \cdot T + C \cdot v^{0{,}16} + D \cdot T \cdot v^{0{,}16}$$
$$A = 13{,}12, B = 0{,}6215, C = -11{,}37, D = 0{,}3965.$$

```
a = 13.12
b = 0.6215
c = -11.37
d = 0.3965

t = float(input('Temperatur: '))
v = float(input('Windgeschwindigkeit: '))

print(
    a +
    b * t +
    c * v**.16 +
    d * t * v**.16
)
```