

Introduction to Python Programming

4 – Variables, Types, Operators

Josef van Genabith (Stefan Thater)
Dept. of Language Science & Technology
Universität des Saarlandes

WS 2022/23



Recap – Calculator

```
print("Please enter the numerator:")
x_str = input()
x = int(x_str)

print("Please enter the denominator:")
y_str = input()
y = int(y_str)

r_mod = x % y
r_div = x / y

print("Result of the division operation:", r_div)
print("Result of the modulo operation:", r_mod)
```

Recap – Calculator (alternatively)

```
print("Please enter the numerator:")  
x = int(input())  
  
print("Please enter the denominator:")  
y = int(input())  
  
print("Result of the division operation:", x / y)  
print("Result of the modulo operation:", x % y)
```

Recap – Calculator (alternatively)

```
x_str = input("Please enter the numerator: ")
x = int(x_str)

y_str = input("Please enter the denominator: ")
y = int(y_str)

print("Result of the division operation:", x / y)
print("Result of the modulo operation:", x % y)
```

Recap

- What is an **algorithm**?
- What is a **program**?
- **Requirements** for algorithms?
- What is **compilation**?
- What is **interpretation**?
- What does **platform independence** mean?

Imperative Programming Paradigm

- Imperative / procedural programming:
 - ▶ „First do this, then do this.“
- Sequence of commands / instructions
- Control structures (loops, branching structures) execute computational steps
- The state of the program changes as a function of time.
- Commands can be grouped into procedures.

Elements of Imperative Programming

- Variables
- Assignments
- Expressions (e.g. numbers, function calls, ...), in general things that evaluate to something ...
- Control Structures: loops, branches, ...

Values and Variables

- An expression is everything that has or evaluates to a value:
 - ▶ Numbers, strings, lists, ...
 - ▶ Sums of numbers, products,, concatenation of strings, ..., function calls, ...
- Variable assignment

```
x = 1          # assign 1 to the variable x
y = x + x      # assign the result of x + x to y
z = [x, y]     # assign a list containing x and y to z
```

- Think of variables as placeholders for values
 - ▶ This is actually not true. Variables are pointers to values. The difference between a placeholder and a pointer will become clear in our next lectures.

Some Data Types

- **Boolean:** `True`, `False`
- **Numbers:** `int (2)`, `float (2.0)`, `complex`
- **Strings:** `str ('Hello')`
- **Collections:** `tuple`, `list`, `set`, `dict`, ...

Converting between Different Data Types

```
>>> x = "17"
>>> x
'17'
>>> type(x)
<class 'str'>
>>> y = int(x)
>>> y
17
>>> type(y)
<class 'int'>
>>> z = str(y)
>>> z
'17'
```

Dynamic Typing

- Variables in Python do not have fixed data types.
- The type of a variable is the assigned value's data type. So typing of a variable happens **when** you assign a value to that variable.
- During runtime, a variable can take values of different types (but this is generally considered **bad style ...**)

```
>>> x = 15.4
>>> type(x)
<type 'float'>
>>> x = "Python is great!"
>>> type(x)
<type 'str'>
```

Floating Point Numbers

- Decimal numbers are represented as floats: 1.1, 47.11
- Range depends on system
- BE CAREFUL! Often, the internal representation of floating point numbers is imprecise.

```
>>> 1.0 - 0.9 == 0.1
False
```

- \Rightarrow use ϵ when comparing floating point numbers

```
>>> x = 1.0
>>> y = 0.9
>>> abs(x - y) < 0.0000000000000001
True
```

Expressions

Expressions = constructs describing a value

We distinguish:

- ▶ **Literals** = expressions from which the value can be directly read off: `1.0`, `True`, `"Python"`, ...
- ▶ **Variables** = references to values
- ▶ **Complex expressions** with operators or function calls: `3+5`, `max([1, 2, 3])`, ... are things that can be evaluated

Elementary Arithmetic Operators

Addition	$a + b$
Subtraction	$a - b$
Multiplication	$a * b$
Division	a / b
Modulo	$a \% b$

```
>>> a = 1
>>> b = 2.4
>>> a + b
3.4
```

- If a and b do not have the same type, the operations result in a value of the **more general** type.
- What are the types in the example? Which type is more general? Why?

Precedence

- Expressions may contain multiple operators: $3 + 2 * 4$
- Precedence = order in which operators are evaluated
- Standard precedence rules: **multiplication / division** before **addition / subtraction**
- Parentheses **indicate/fix** precedence directly
- Style: sometimes it is recommended to use parentheses even if they are not strictly necessary (legibility)
- Don't use parentheses when precedence is obvious or irrelevant: $2+3+4$ is considered better than $2+(3+4)$

```
>>> 3 + 2 * 4
```

```
11
```

```
>>> (3 + 2) * 4
```

```
20
```

Boolean (Truth Values)

- The type `bool` represents the two truth values `True` and `False`
- Homework: refresh your knowledge on truth tables
- Precedence: `not > and > or`
 - ▶ `a and not b or c = (a and (not b)) or c`
- Short-circuit evaluation: the evaluation stops as soon as the result is evident (`True or ...`)

Negation	<code>not a</code>
Conjunction	<code>a and b</code>
Disjunction	<code>a or b</code>

Boolean (Truth Values)

- The type `bool` represents the two truth values `True` and `False`
- Homework: refresh your knowledge on truth tables

Negation	<code>not a</code>
Conjunction	<code>a and b</code>
Disjunction	<code>a or b</code>

- Precedence: `not` > `and` > `or`
 - ▶ `a and not b or c = (a and (not b)) or c`
- Short-circuit evaluation: the evaluation stops as soon as the result is evident (`True or ...`)

Logic: Connectives

AND			OR			NOT	
P	Q	(P ^ Q)	P	Q	(P v Q)	P	~P
T	T	T	T	T	T	T	F
T	F	F	T	F	T	F	T
F	T	F	F	T	T		
F	F	F	F	F	F		

IMPLIES			IF AND ONLY IF		
P	Q	(P => Q)	P	Q	(P <=> Q)
T	T	T	T	T	T
T	F	F	T	F	F
F	T	T	F	T	F
F	F	T	F	F	T

Boolean (Truth Values)

Negation	not a
Conjunction	a and b
Disjunction	a or b

Logic: Connectives

AND

P	Q	$(P \wedge Q)$
T	T	T
T	F	F
F	T	F
F	F	F

OR

P	Q	$(P \vee Q)$
T	T	T
T	F	T
F	T	T
F	F	F

NOT

P	$\neg P$
T	F
F	T

IMPLIES

P	Q	$(P \Rightarrow Q)$
T	T	T
T	F	F
F	T	T
F	F	T

IF AND ONLY IF

P	Q	$(P \Leftrightarrow Q)$
T	T	T
T	F	F
F	T	F
F	F	T

String Literals

- Strings are sequences of characters
 - ▶ (no separate type for characters, i.e. characters are strings ...).
- Examples
 - ▶ `"This is a string."`
 - ▶ `'That, too.'`
 - ▶ `"He said \"Hello\"."`
 - ▶ `'He said "Hello".'`
- If you want to use an encoding other than ASCII or UTF8, specify the encoding in the first code line:

```
# -*- coding: latin-1 -*-
```

Lists

- Lists are collections of values, e.g., `[1, 'two', 3.0]`
- Access to the *n*th item: `somelist[n]`

```
>>> days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
>>> days[0] # first element
'Mon'
>>> days[6] # 7th element
'Sun'
>>> days[-1] # last element
'Sun'
```

Exercise – What are the types?

- (1) 1.0
- (2) "a"
- (3) False
- (4) 5
- (5) ['hello', 'world']
- (6) "
- (7) "Python:"
- (8) [2.5, 6.7, 1.2, 4]
- (9) "7"

Some String Operators

- Concatenation:
 - ▶ `'Hello' + 'World' ⇒ 'HelloWorld'`
- Access to individual characters with list indices:
 - ▶ `'Hello'[0] ⇒ 'H'`
 - ▶ `'Hello'[1] ⇒ 'e'`
- Test whether a substring occurs:
 - ▶ `'ell' in 'Hello' ⇒ True`
 - ▶ `'lle' in 'Hello' ⇒ False`
- Length: `len('Hello') ⇒ 5`

Relational Operators

less than	$a < b$
greater than	$a > b$
less than or equal to	$a \leq b$
greater than or equal to	$a \geq b$
Equal to	$a == b$
not equal to	$a \neq b$

- Result of a comparison: Boolean
- True or False

Variables

- Placeholders for values
- One can assign the value of an expression to variables
- Variables can be evaluated in order to use their value in an expression
- `print()` is a function that prints the value of an expression to the console (the standard output)

```
>>> number = 123
>>> number = number + 2
>>> number
125
>>> print(number)
125
```


Variables

- Variables must start with a **letter** or “_”. The remainder may include digits.
- Umlauts etc. are allowed in Python 3, but we recommend to stick to ASCII any ways!
- The name of a variable must not be a keyword (**if**, **while**, **and** etc.)
- The names are case-sensitive
 - ▶ **x** and **X** are different variables!
 - ▶ Convention: variables should start with a lower-case letter

Which ones are allowed?

```
foo
2foo
foo2
_foo
if
überzweig
```

Assignments

- `var = expr`
 - ▶ the expression `expr` is evaluated, then its value is assigned to the variable `var`.
- `var1 = ... = varn = expr`
 - ▶ the value of `expr` is assigned to all variables `vari`
- `var1, ..., varn = expr1, ..., exprn`
 - ▶ all `expri` are evaluated, then the corresponding values are assigned to `vari`

```
>>> x = 1
>>> y = z = 2
>>> u, w = 3, 4
>>> u, w = w, u
```

Assignments

Long form	shorthand
<code>x = x + expr</code>	<code>x += expr</code>
<code>x = x - expr</code>	<code>x -= expr</code>
<code>x = x * expr</code>	<code>x *= expr</code>
<code>x = x / expr</code>	<code>x /= expr</code>
<code>x = x % expr</code>	<code>x %= expr</code>

```
>>> x = 5
>>> x += 3
>>> print(x)
8
>>> x /= 2
>>> print(x)
4
```

Exercise

(a) `a, b = 5, 3`

(b) `c = 'test'`

(c) `a = a % b`

(d) `b *= b`

(e) `a = c[1]`

(f) `c = a = b = True`

(g) `c = not(a or b)`

(h) `a = b or c`

(i) `b = str(a) + " Love"`

The listing shows several steps of a program. For each step, write down the **values** and **types** of `a`, `b` and `c`.

Example: read text from console

```
first = input("Input your first name: ")
last = input("Input your last name: ")
middle = input("Input your middle name: ")
print("Your full name is: ", first, middle, last)
```

- `variable = input(prompt)`
 - ▶ prints the text given in `prompt` on the console (terminal, standard input)
 - ▶ waits for the user to enter some text string (terminated by pressing the 'enter' key)
 - ▶ assigns the text string that the user has entered to the variable

How to debug?

- (in a simple way, to start with)
- Don't write down the entire program at once.
Test after each line:
 - ▶ Are there any syntax errors? IDLE shows them immediately
 - ▶ **Print out the value of the important variables and check whether they are what you expect!**
 - ▶ After testing the line, comment out or remove the print statement.

