JÖNKÖPING UNIVERSITY

School of Engineering

BASICS IN PYTHON, PART 1

Peter Larsson-Green

Jönköping University

Autumn 2018

PETER'S CONVENTION

```
This box represents Python code written in plain text.
```

This box represents an entire program
which consists of a sequence of statements.

This box represents a single statement.

This box represents a single expession.

The arrow → represents an evaluation step of some kind, e.g.:

3 + 8 → 11

IMPERATIVE PROGRAMS

A program consists of:

- A sequence of statements.

A statement consists of:

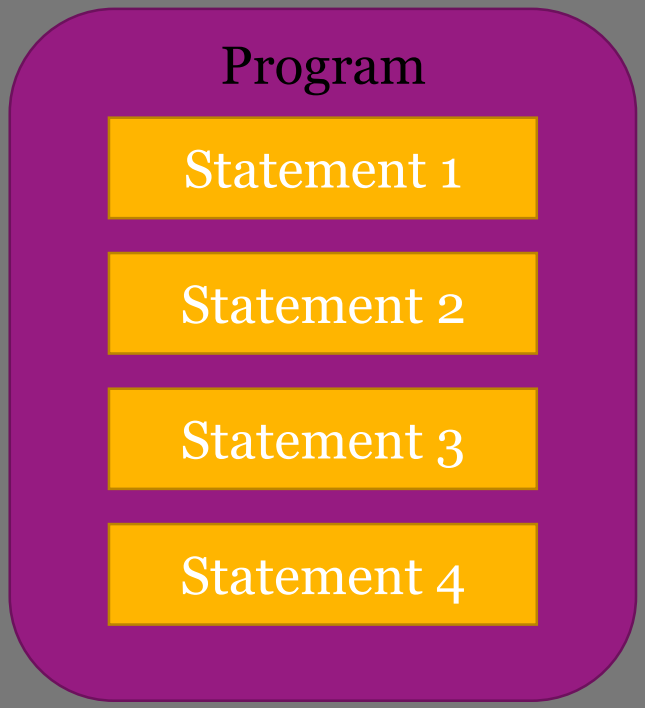
- Other statements and expressions.

Expressions evaluate to:

- Values.

Executed statements:

- Alters the state of the program.



Name	Value
x	12
y	36

Variable table.

EXPRESSIONS

- Is something your computer can evaluate.
- It will always evaluate to a value.
- Many programming languages have a tool with an REP-loop.
  - Read the entered expression.
  - Evaluate it.
  - Print the computed value.

INTEGERS

- Used to represent "how many" (something countable).
- An expression consisting of a sequence of digits is evaluated to a integer.

Examples

37 37 → 37

52 52 → 52

FLOATS

- Used to represent "how much".
- An expression consisting of a sequence of "digits and one decimal dot" is evaluated to a float.

Examples

`37.5` `37.5` → `37.5`

`52.0` `52.0` → `52.0`

THE NEGATION EXPRESSION

Syntax: `- <expr>`

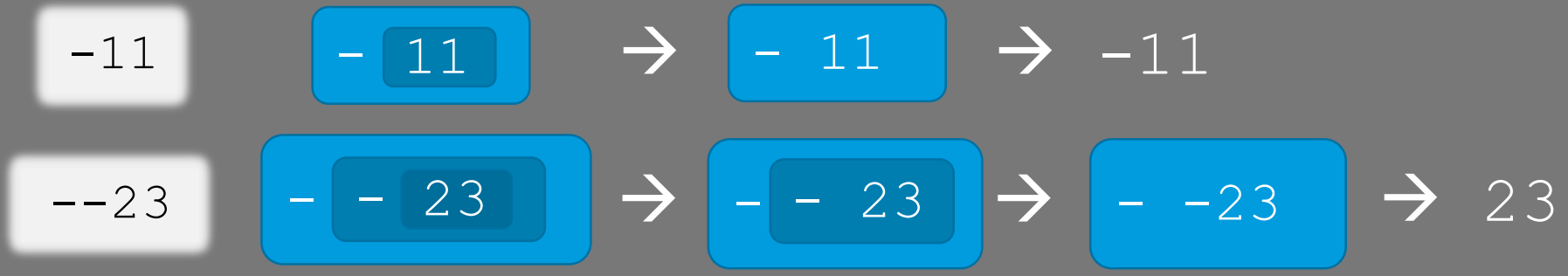
How it is computed

1. Evaluate `<expr>`.
2. Negate that value.

In reality, `-0` and `0` represents the same number.

How about in computers?

Examples



BINARY MATHEMATICAL EXPRESSIONS

Syntax:

```
<expr1> <operator> <expr2>
```

How it is computed

1. Evaluate `<expr1>`.
2. Evaluate `<expr2>`.
3. Apply `<operator>` on the computed values.

Examples

`5+3`	`5 + 3` →	`5 + 3` →	`5 + 3` →	`8`
`4-7`	`4 - 7` →	`4 - 7` →	`4 - 7` →	`-3`
`3*2`	`3 * 2` →	`3 * 2` →	`3 * 2` →	`6`
`1/4`	`1 / 4` →	`1 / 4` →	`1 / 4` →	`0.25`

What about division by zero?

BINARY MATHEMATICAL EXPRESSIONS

Examples

WHAT ABOUT MORE OPERANDS?

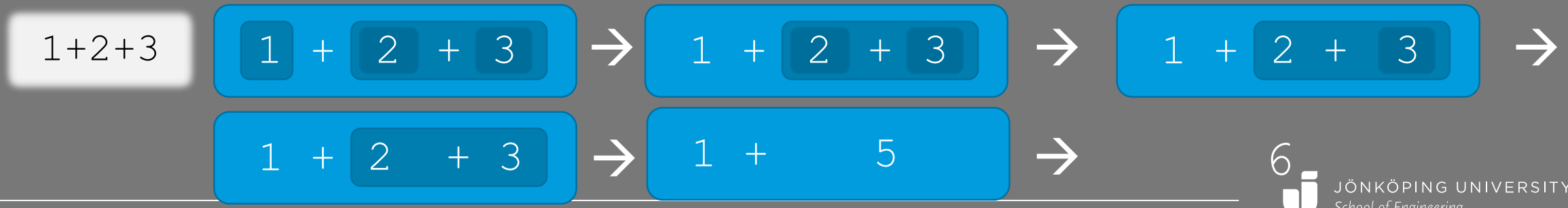
Syntax: `<expr1> <operator> <expr2>`

- `<expr1>` and `<expr2>` can in turn be binary mathematical expressions!

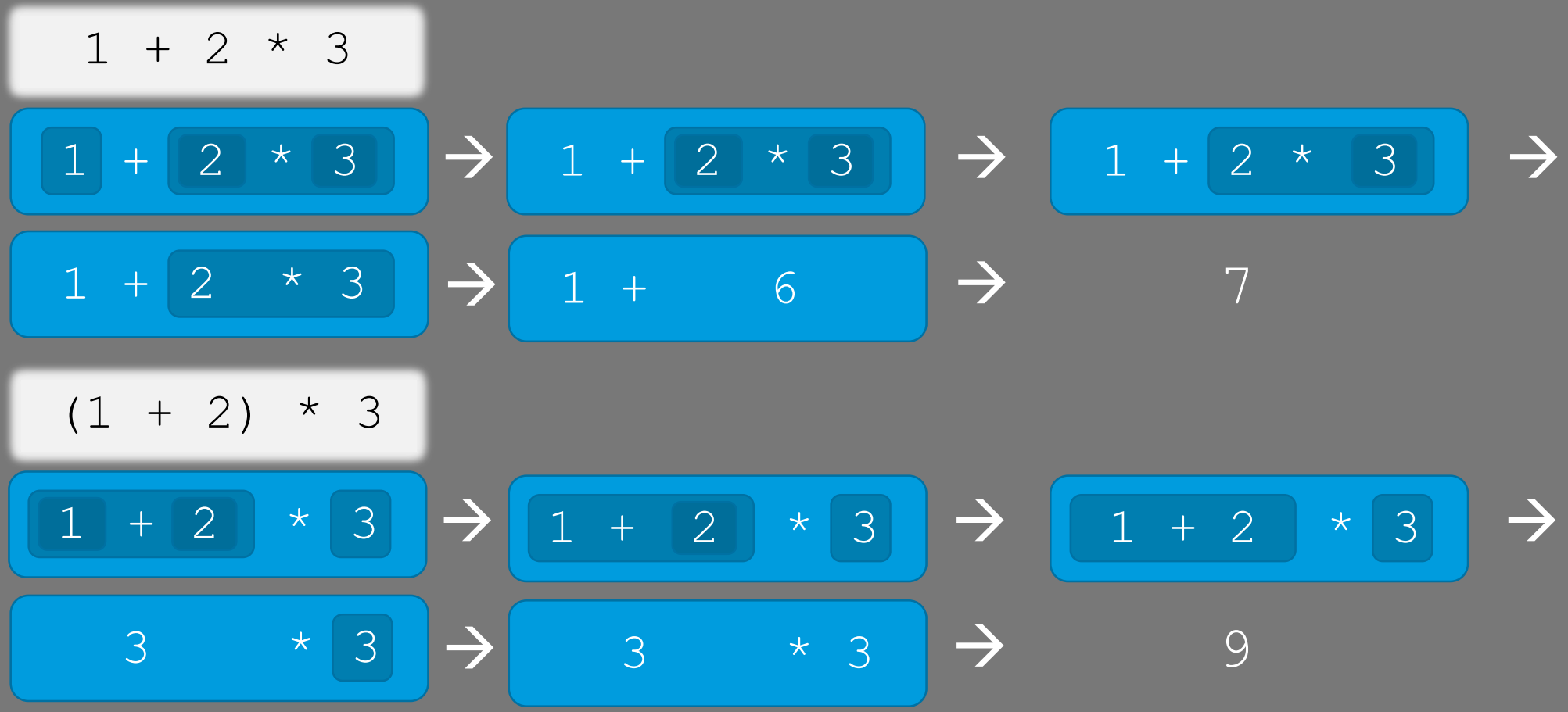
`<expr1> <op> <expr2> <op> <expr3>`

`<expr1> <op> <expr2> <op> <expr3>`

- Example



WHAT ABOUT MORE OPERANDS?



Ordinary math operator precedence applies!

EXAMPLE

To calculate the size of the surface of a sphere, the following formula is used:

$$Area = 4 * \pi * r * r$$

$$\pi = 3.14159\ldots$$

The radius of Earth is about 6,371 kilometers.

- How big is the surface of Earth?
  - `4*3.14159*6371*6371` kilometer².
  - `4*3.14159*6371000*6371000` meter².
- How big would it be if the radius is 500 kilometers shorter?
  - `4*3.14159*(6371-500)*(6371-500)` kilometer².
  - `4*3.14159*(6371000-500000)*(6371000-500000)` meter².

STATEMENTS

- Most high level programs consist of a sequence of statements.
- They are executed from top to bottom.
- An executed statement results in side effects. For example…
  - Storing a value in a variable.
  - Conditionally execute other statements.
  - Conditionally repeat execution of other statements.

VARIABLES

A variable is a sequence of characters storing a value.

- Statement creating a variable: `<variable> = <expr>`

How it is executed

1. Evaluate `<expr>` to a value.
2. Create a variable named `<variable>` and store the value there.

Name	Value
`<variable>`	`The value`

Variable table.

`<variable>` → `The value`

EXAMPLE

- Calculate how big the surface of Earth is (`4*3.14159*6371*6371`).

```
pi = 3.14159
earthRadius = 6371
earthArea = 4*pi*earthRadius*earthRadius
```

Name	Value
pi	3.14159
earthRadius	6371
earthArea	510064041.08

Variable table.

Benefits

- Code is easier to read.
- Do not duplicate the hard coded radius value.

VARIABLES NAMING CONVENTION

The name of the variable should reflect the value it stores.

- Makes the code easier to read.

```
x = 5
```

```
z = 7.23
```

```
numberOfStudents = 5
```

```
amountOfWater = 7.23
```

- Common naming conventions:
  - `writeItLikeThis` (camelCase, first letter lowercased)
  - `WriteItLikeThis` (camelCase, first letter capitalized)
  - `write_it_like_this` (used in Python!)

REASSIGNMENT STATEMENT

Stores a new value in an existing variable.

Syntax: `<variable> = <expr>`

How it is executed

1. Evaluate `<expr>`.
2. Store the evaluated expression in the variable named `<variable>`.

EXAMPLE

Purpose: to compute the sum of the integers between 0 and 2.

```
sum = 0
sum = sum + 1
sum = sum + 2
```

Name	Value
sum	0 1 3

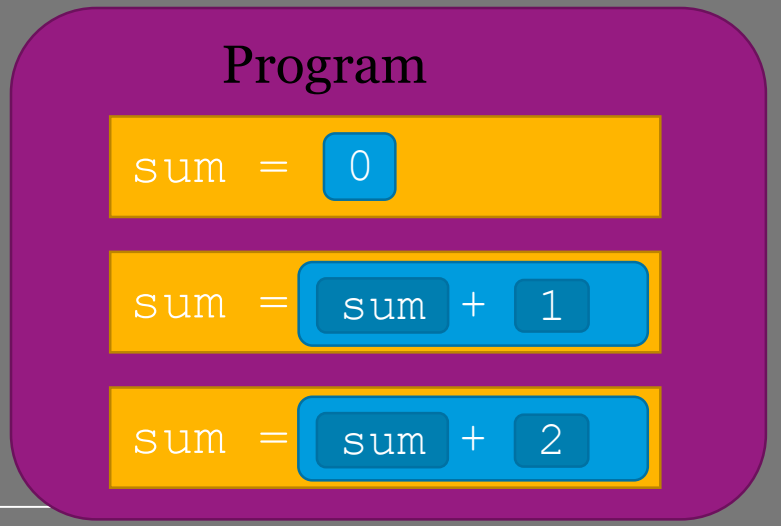
Variable table.

How many statements?

- 3

How many expressions?

- 7



EXAMPLE

Purpose: to compute the sum of the integers between 0 and 2.

```
# Let's be a computer and execute the statements!
sum = 0
sum = sum + 1
sum = sum + 2
```

EXAMPLE

Purpose: to compute the sum of the integers between 0 and 2.

```
sum = 0 # I create the variable sum, storing: 0 → 0.
sum = sum + 1
sum = sum + 2
```

EXAMPLE

Purpose: to compute the sum of the integers between 0 and 2.

```
sum = 0
sum = sum + 1 # I store a new value in sum: sum+1 → 0+1 → 1.
sum = sum + 2
```

EXAMPLE

Purpose: to compute the sum of the integers between 0 and 2.

```
sum = 0
sum = sum + 1
sum = sum + 2 # I store a new value in sum: sum+2 → 1+2 → 3.
```

EXAMPLE

Purpose: to compute the sum of the integers between 0 and 2.

```
sum = 0
sum = sum + 1
sum = sum + 2
# And I'm done!
```

STRINGS

Represents a sequence of characters.

- Expressions creating strings:

```
"This is a string."  → This is a string.
'This is a string.'  → This is a string.
```

- Escaped characters have special meaning:
  - `\"` = "
  - `\'` = '
  - `\n` = newline

```
"This is\na string."  → This is
                        a string.
```

STRINGS

- Multiline strings:

```
"""This is a
string covering
multiple lines."""
```

→

```
This is a
string covering
multiple lines.
```

  - Works with `'''` as well.
- The + operator can be used to concatenate strings:

```
"This is " + 'a string!'
```

→ `This is a string!`

INPUT OUTPUT

Printing a value to the console:

```
print("The thing to print!")
print(52)
```

```
The thing to print!
52
```

Reading a string from the console:

```
input()
```

```
Hello!
```

```
input("Enter something: ")
```

```
Enter something: Hello!
```

```
entered_text = input()
```

EXAMPLE

```
name = input("Enter your name: ")
print("Hello "+name+"!")
```

```
Enter your name: Peter
Hello Peter!
```

```
age = input("Enter your age: ")
print("Your age doubled is: "+age*2)
```

```
Enter your age: 2
Your age doubled is: 22
```

```
age = input("Enter your age: ")
age = int(age)
print("Your age doubled is: "+age*2)
```

```
Enter your age: 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

EXAMPLE

```
age = input("Enter your age: ")
age = int(age)
print("Your age doubled is: "+str(age*2))
```

```
Enter your age: 2
Your age doubled is: 4
```

```
Enter your age: a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int()
with base 10: 'a'
```

DIFFERENT TYPE OF ERRORS

- Syntax errors
  - What you have written is not valid Python code.
  - Example: `my_variable = 4 + 6 -`
  - Python do not understand what you want → nothing will be executed.
- Runtime errors
  - Python discovers the error while executing your code.
  - Example: `my_variable = 4 / 0`
- Logical errors
  - Python runs your entire program, but it does not work as you want.
  - Example: `average = 4 + 6 / 2`