ILP 2022 – W1S2
Variables, math operators, comments, printing and getting

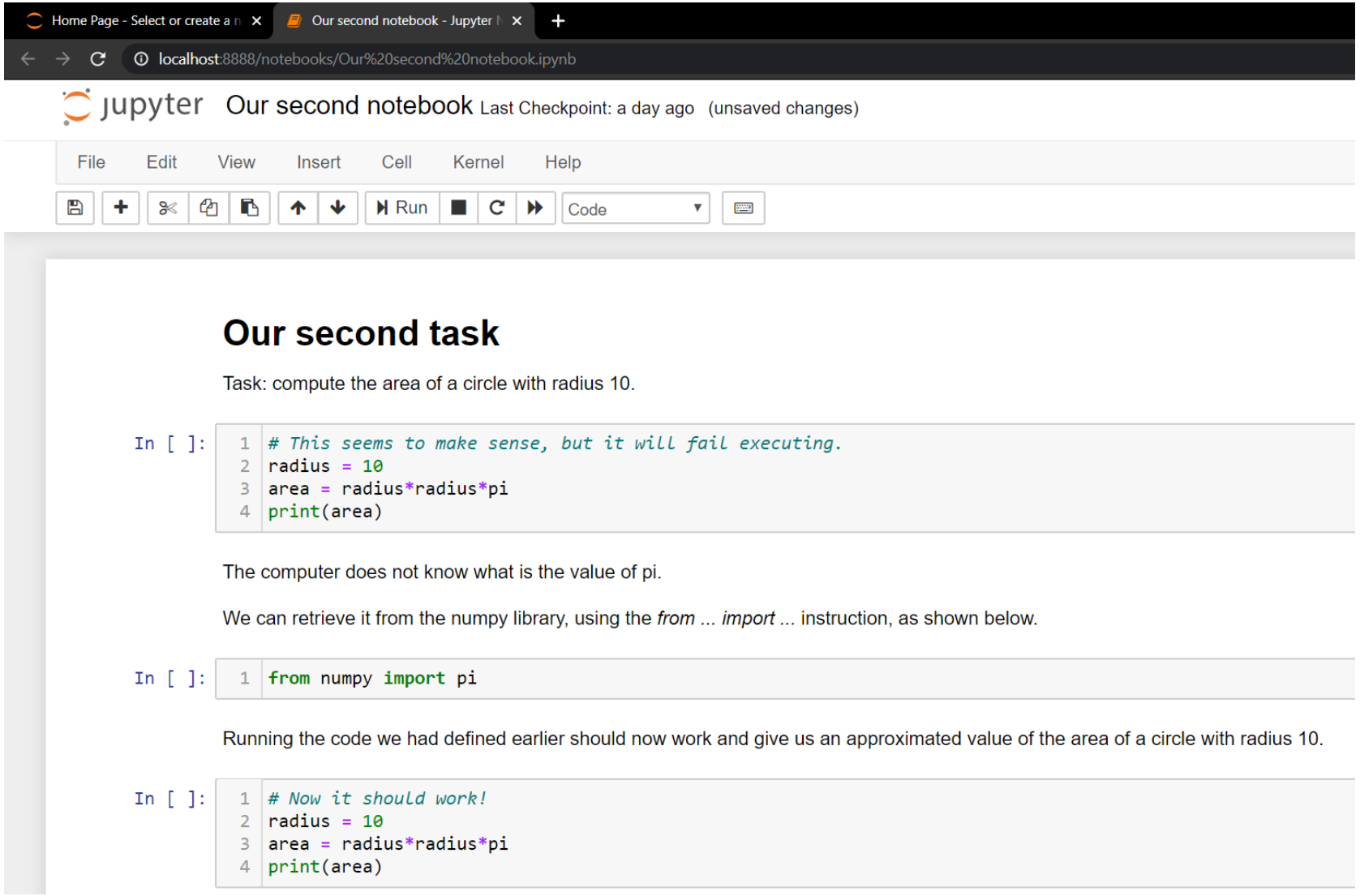
Matthieu DE MARI – Singapore University of Technology and Design

Outline (Week1, Session2 – W1S2)

- **Variables:** Naming conventions, types, IDs, assignments, conversions
- **Basic math operators in Python:** +, -, *, /, //, %, **
- **Combining operators and assignments**
- **Commenting:** in-line, block, docstring and header comments
- **Printing and getting:** displaying with print(), getting with input()

- **Bonus (out-of-class, if time allows!):** what happens backstage when I run my code?

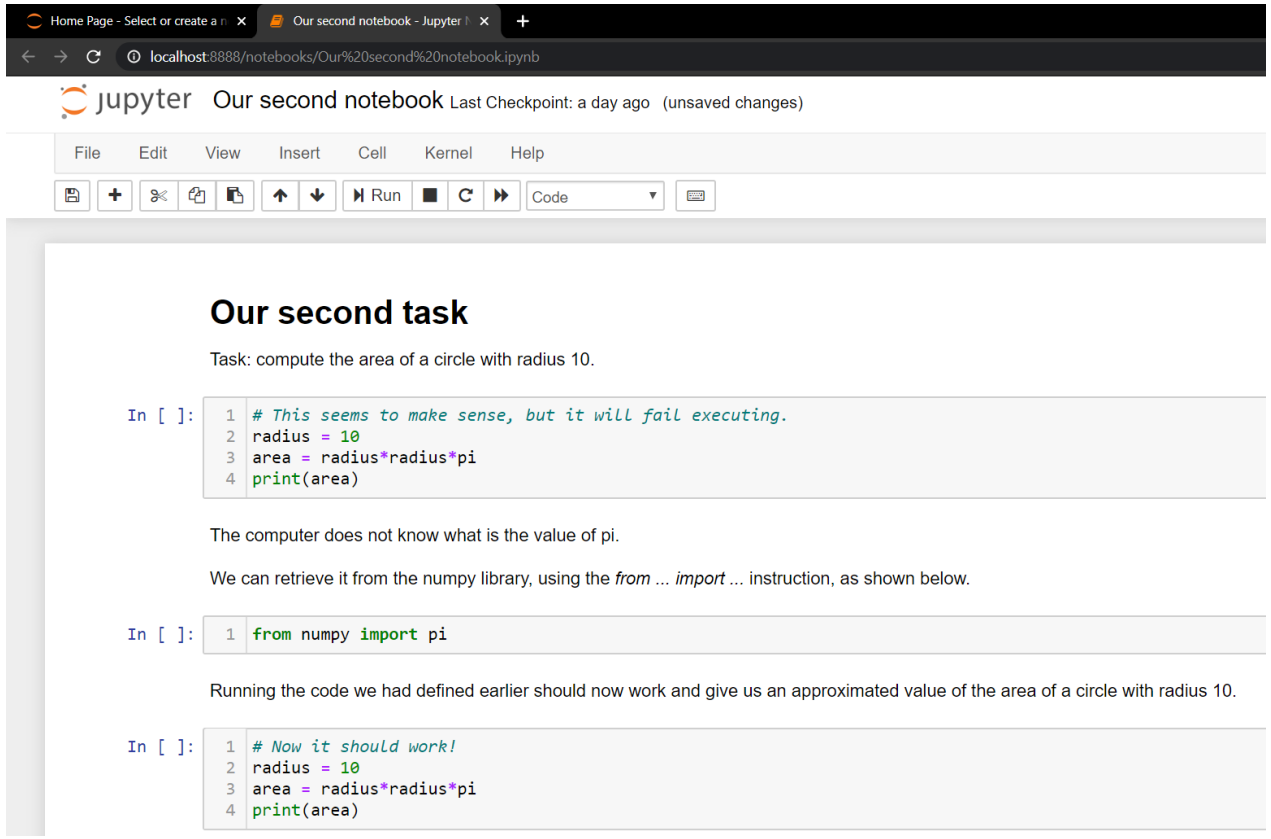
Recall: our second task



Recall: our second task

In this task, we have intuitively used a few concepts, such as:

1. **Assigning values to memory (variables)**
2. **Using basic math operations**
3. **Commenting your code**
4. **Printing**
5. **Executing a coding script in console or an IDE**
6. **(Importing extra stuff)**



→ In the next couple of slides, we will go deeper into these concepts

About variables names

- **Variables names** can include any combination of **letters** (both lowercase and uppercase), **digits** and **underscore symbols** (_).
- You can use the **underscore** symbol as a separator to make variables names a bit more explicit.

```
1  # A variable name that is not explicit
2  x = 10
3
4  # A variable name that is more explicit
5  radius = 10
6
7  # A variable name that is, maybe, a bit too explicit?
8  number_of_students_in_ILP_summer_school = "THERE'S TOO MANY OF THEM!"
```

Matt's Great advice #2

Matt's Great Advice #2: make your variables names explicit.

It is a good idea to make your variables names **explicit**, rather than using meaningless ones that leave your reader guessing (x, y, a, b1, c2, etc.).

You can **use underscores** if you find it helpful (e.g. current_year = 2020).

Your future self, and colleagues, will thank you, when they work on your code later on.

About variables names

- **Variables names** can include any combination of **letters** (both lowercase and uppercase), **digits** and **underscore symbols** (_).

However,

- They cannot **start with a digit**, or use **special characters other than underscores**.
- They may also not use some **reserved keywords** (e.g. import, print, etc.)

```
# This is okay as a variable name
a_1st_VaRiAbLe = 1
```

```
# Variables cannot start with a digit
2nd_variable = 2
```

```
  File "<ipython-input-3-f01f45c0ae38>", line 2
    2nd_variable = 2
     ^
SyntaxError: invalid syntax
```

```
# But they can start with an underscore (_)
_third_variable = 3
```

```
# In fact, this is also a valide name!
# (But not explicit at all!)
_ = 4
print(_)
```

```
4
```

```
# Variables cannot use special symbols,
# except for the underscore symbol (_)
v@r!aBl€ = 5
```

```
  File "<ipython-input-8-f47b51204aa7>", line 2
    v@r!aBl€ = 5
      ^
SyntaxError: invalid syntax
```

```
# Cannot use keywords as variable names
import = 6
```

```
  File "<ipython-input-20-4526a1d99ee1>", line 2
    import = 6
           ^
SyntaxError: invalid syntax
```

Multiple assignments

Python also allows **multiple assignments** using

1. **Successive multiple equal signs:** the value on the far right is assigned to all variables names on the left side of an equal sign.

Multiple assignments

```
1 # A multiple assignment
2 var1 = var2 = var3 = 10
3 print(var1)
4 print(var2)
5 print(var3)
```

```
10
10
10
```

Multiple assignments

Python also allows **multiple assignments** using

1. **Successive multiple equal signs:** the value on the far right is assigned to all variables names on the left side of an equal sign.
2. **Or by having several variables names and values separated by commas on both sides:** the values are respectively assigned to the variables names.

Multiple assignments

```
1  # A multiple assignment
2  var1 = var2 = var3 = 10
3  print(var1)
4  print(var2)
5  print(var3)
```

```
10
10
10
```

```
1  # Another multiple assignment
2  var4, var5 = 8, "Some text"
3  print(var4)
4  print(var5)
```

```
8
Some text
```

Variable types

- **Definition (variable type/class):** Everything is stored in memory as 0s and 1s. A **variable type/class** defines how the sequence of 0s and 1s should be interpreted.

Variable types

- **Definition (variable type/class):** Everything is stored in memory as 0s and 1s. A **variable type/class** defines how the sequence of 0s and 1s should be interpreted.

Let us start with three basic types

- **int:** an integer number.
- **float:** a floating point number, for decimal numbers.
- **str:** a string of text.

Variable types

```
1 radius = 10
2 pi = 3.14
3 message = "Hello World!"
```

```
1 print(type(radius))
```

```
<class 'int'>
```

```
1 print(type(pi))
```

```
<class 'float'>
```

```
1 print(type(message))
```

```
<class 'str'>
```

Variable IDs

- **Definition (variable ID):** Everything is stored in memory as 0s and 1s. A **variable ID** returns an integer number, corresponding to a memory address where the variable is being stored in memory.

This is automatically handled by your computer. More on this later, so let us keep it in mind.

Variable IDs

```
1 radius = 10
2 pi = 3.14
3 message = "Hello World!"
```

```
1 print(id(radius))
```

```
140721979856832
```

```
1 print(id(pi))
```

```
2367280141872
```

```
1 print(id(message))
```

```
2367280157744
```

Type conversion

- **Int/Float -> String:** You can convert any int/float to a string using the **str()** function.

Variables conversion

```
1 a_number_as_int = 1024
2 same_number_as_string = str(a_number_as_int)
3 print(a_number_as_int)
4 print(type(a_number_as_int))
5 print(same_number_as_string)
6 print(type(same_number_as_string))
```

```
1024
<class 'int'>
1024
<class 'str'>
```

```
1 a_number_as_float = 1.5
2 same_number_as_string = str(a_number_as_float)
3 print(a_number_as_float)
4 print(type(a_number_as_float))
5 print(same_number_as_string)
6 print(type(same_number_as_string))
```

```
1.5
<class 'float'>
1.5
<class 'str'>
```

Type conversion

- **String -> Int/Float:** You can convert a string of digits to an int/float using **int()** or **float().**
- **Note:** You can convert a string of digits with a single decimal point to a float.
- **Also:** if the string includes a decimal point, it cannot be converted to int.
- **Important note:** if the string contains non-digits characters, the conversion fails.

```
1 number_as_string = "1024"
2 same_number_as_int = int(number_as_string)
3 same_number_as_float = float(number_as_string)
4 print(number_as_string)
5 print(type(number_as_string))
6 print(same_number_as_int)
7 print(type(same_number_as_int))
8 print(same_number_as_float)
9 print(type(same_number_as_float))
```

```
1024
<class 'str'>
1024
<class 'int'>
1024.0
<class 'float'>
```

Type conversion

- **Int <-> Float:** You can convert a float to an int and vice versa.

Note that,

- Converting from int to float adds a single 0 decimal.
- Converting from float to int, will **drop all decimals** (i.e. a truncation of all decimals).

```
1 an_int_number = 12
2 int_to_float_number = float(an_int_number)
3 print(int_to_float_number)
```

```
12.0
```

```
1 a_float_number = 1.8
2 float_to_int_number = int(a_float_number)
3 print(float_to_int_number)
```

```
1
```

Python basic operators

By default, Python comes with a few **basic math operators, for number (int/float) variables**.

- **+:** addition
- **-:** subtraction
- ***:** multiplication

Basic math operators

```
1 # Two numbers
2 a, b = 5, 2
3 print(a)
4 print(b)
```

```
5
2
```

```
1 # Addition
2 print(a+b)
```

```
7
```

```
1 # Subtraction
2 print(a-b)
```

```
3
```

```
1 # Multiplication
2 print(a*b)
```

```
10
```

Python basic operators

By default, Python comes with a few **basic math operators, for number (int/float) variables**.

- **+:** addition
- **-:** subtraction
- ***:** multiplication
- **/:** division
- **//:** integer division
- **%:** remaining of the integer division (a.k.a. modulus)
- ****:** exponentiation

Basic math operators

```
# Two numbers
a, b = 5, 2
print(a)
print(b)
```

```
5
2
```

```
# Addition
print(a+b)
```

```
7
```

```
# Subtraction
print(a-b)
```

```
3
```

```
# Multiplication
print(a*b)
```

```
10
```

```
# Division
print(a/b)
```

```
2.5
```

```
# Floor division
print(a//b)
```

```
2
```

```
# Modulus
print(a%b)
```

```
1
```

```
# Exponentiation
print(a**b)
```

```
25
```

Python basic operators

By default, Python comes with a few **basic math operators, for number (int/float) variables**.

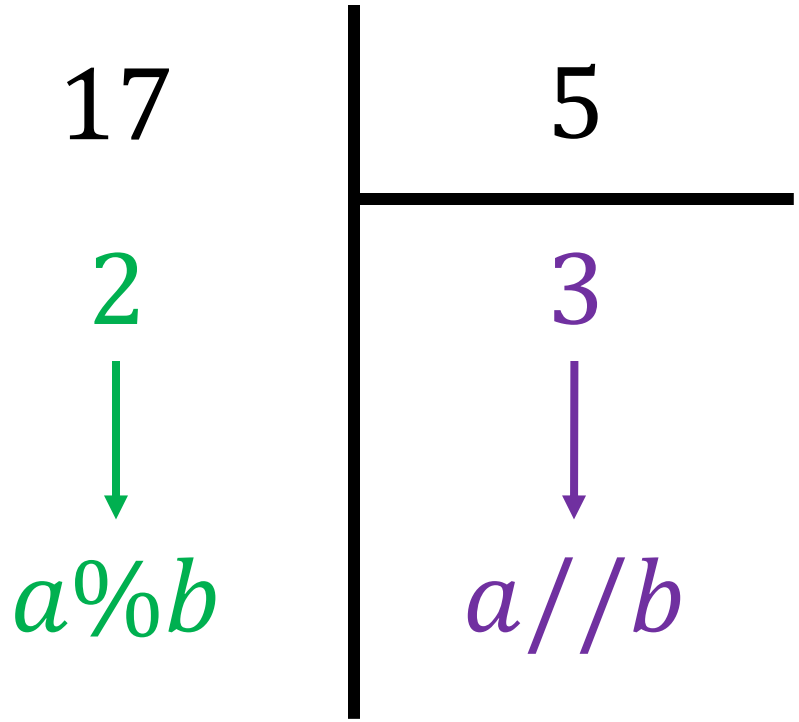
- **+:** addition
- **-:** subtraction
- ***:** multiplication
- **/:** division
- **//:** integer division
- **%:** remaining of the integer division (a.k.a. modulus)
- ****:** exponentiation

```
1  # Floor division
2  print(a//b)
```

```
2
```

```
1  # Modulus
2  print(a%b)
```

```
1
```



Python basic operators

By default, Python comes with a few **basic math operators, for number (int/float) variables**.

- **+:** addition
- **-:** subtraction
- ***:** multiplication
- **/:** division
- **//:** integer division
- **%:** remaining of the integer division (a.k.a. modulus)
- ****:** exponentiation

Basic math operators

```
# Two numbers
a, b = 5, 2
print(a)
print(b)
```

```
5
2
```

```
# Addition
print(a+b)
```

```
7
```

```
# Subtraction
print(a-b)
```

```
3
```

```
# Multiplication
print(a*b)
```

```
10
```

```
# Division
print(a/b)
```

```
2.5
```

```
# Floor division
print(a//b)
```

```
2
```

```
# Modulus
print(a%b)
```

```
1
```

```
# Exponentiation
print(a**b)
```

```
25
```

Python basic operators

By default, Python comes with a few **basic math operators, for number (int/float) variables**.

- **+:** addition
- **-:** subtraction
- ***:** multiplication
- **/:** division
- **//:** integer division
- **%:** remaining of the integer division (a.k.a. modulus)
- ****:** exponentiation

Basic math operators

```
# Two numbers
a, b = 5, 2
print(a)
print(b)
```

```
5
2
```

```
# Addition
print(a+b)
```

```
7
```

```
# Subtraction
print(a-b)
```

```
3
```

```
# Multiplication
print(a*b)
```

```
10
```

```
# Division
print(a/b)
```

```
2.5
```

```
# Floor division
print(a//b)
```

```
2
```

```
# Modulus
print(a%b)
```

```
1
```

```
# Exponentiation
print(a**b)
```

```
25
```

Python basic operators

By default, Python comes with a few **basic math operators, for number (int/float) variables**.

- **+:** addition
- **-:** subtraction
- ***:** multiplication
- **/:** division
- **//:** integer division
- **%:** remaining of the integer division (a.k.a. modulus)
- ****:** exponentiation

Watch out for the types of variables!

```
1 a = 2
2 b = "Hello World!"
3 print(a+b)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-4eaf3ef22f17> in <module>
      1 a = 2
      2 b = "Hello World!"
----> 3 print(a+b)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
TypeError: unsupported operand type(s)
for +: 'int' and 'str'
```

Combining operators and assignments

You can mix both operations and assignments.

- For instance, the **operation +=** sums values and the right- and left-hand sides, and assigns the result to the variable on the left-hand side.

- Similarly, we have: **-=, *=, /=, //=, %=, **=.**

Combining math operations and assignments

```
1  # Addition
2  a, b = 5, 2
3  print(a)
4  a += b
5  print(a)
```

```
5
7
```

```
1  # Multiplication
2  a, b = 5, 2
3  print(a)
4  a *= b
5  print(a)
```

```
5
10
```

Operation precedence

By default, operations are computed **from left to right**.

However, **operation precedence** applies: exponentiations before multiplications/divisions, before additions/subtractions.

As in mathematics, **parentheses** can be used to “force” precedence.

Operation precedence

```
1  # Operation precedence
2  a, b, c = 4, 3, 2
3  result = a+b*c
4  print(result)
```

```
10
```

```
1  # Operation precedence
2  a, b, c = 4, 3, 2
3  result = (a+b)*c
4  print(result)
```

```
14
```

```
1  # Operation precedence
2  a, b, c = 4, 3, 2
3  result = a*b**c
4  print(result)
```

```
36
```

Activity 1: compute the roots of a quadratic equation, with strictly positive determinant

- Consider the quadratic equation
$$ax^2 + bx + c = 0$$
- With $a, b, c \in \mathbb{R}$, such that the determinant $\Delta > 0$
$$\Delta = b^2 - 4ac$$
- The two roots of this equation (x_1, x_2) are given by

$$x_1 = \frac{-b + \sqrt{\Delta}}{2a} \quad and \quad x_2 = \frac{-b - \sqrt{\Delta}}{2a}$$

Activity 1: compute the roots of a quadratic equation, with strictly positive determinant

- Consider the quadratic equation
$$ax^2 + bx + c = 0$$
- With $a, b, c \in \mathbb{R}$, such that the determinant $\Delta > 0$
$$\Delta = b^2 - 4ac$$
- The two roots of this equation (x_1, x_2) are given by
$$x_1 = \frac{-b + \sqrt{\Delta}}{2a} \quad and \quad x_2 = \frac{-b - \sqrt{\Delta}}{2a}$$

- **Task:** Consider a quadratic equation with the following values: $\boldsymbol{a = 2}$, $\boldsymbol{b = -2}$ and $\boldsymbol{c = -24}$.

Write a Python program

1. that **computes** the value of Δ and **prints** it on screen (to check it is strictly positive)
2. And, later on, **computes** the values of the roots x_1 and x_2 and **prints** them on screen.

Activity 1: compute the roots of a quadratic equation, with strictly positive determinant

- For this activity, you may use the notebook

  Activity 1 - computing the roots of a quadratic equation.ipynb

- Remember to start a notebook environment using the command below, in a console, with location appropriately selected

  py –m notebook

- Also, remember, the **square root of x** is equivalent to **x being exponentiated to the power 0.5 (or import numpy sqrt function!)**

$$\sqrt{x} = x^{0.5} = x ** (0.5)$$

Activity 1: compute the roots of a quadratic equation, with strictly positive determinant

- Consider the quadratic equation
$$ax^2 + bx + c = 0$$
- With $a, b, c \in \mathbb{R}$, such that the determinant $\Delta > 0$
$$\Delta = b^2 - 4ac$$
- The two roots of this equation (x_1, x_2) are given by
$$x_1 = \frac{-b + \sqrt{\Delta}}{2a} \quad and \quad x_2 = \frac{-b - \sqrt{\Delta}}{2a}$$

- **Task:** Consider a quadratic equation with the following values: $\boldsymbol{a = 2}$, $\boldsymbol{b = -2}$ and $\boldsymbol{c = -24}$.

Write a Python program

1. that **computes** the value of Δ and **prints** it on screen (to check it is indeed strictly positive)
2. And, later on, **computes** the values of the roots x_1 and x_2 and **prints** them on screen.

→ **Expected answers: $\boldsymbol{\Delta = 196, x_1 = 4, x_2 = -3.}$**

Activity 1: compute the roots of a quadratic equation, with strictly positive determinant

```python
# 1. Define the coefficients (a,b,c) of the quadratic equation
a = 2
b = -2
c = -24

# 2. Compute the discriminant delta
delta = b**2 - 4*a*c

# 3.Print delta to check it is strictly positive
print(delta)

# 4. Compute the two roots (x1, x2)
x1 = (-b + delta**0.5)/(2*a)
x2 = (-b - delta**0.5)/(2*a)

# 5. Print the roots
print(x1)
print(x2)
```

```
196
4.0
-3.0
```

Activity 1: compute the roots of a quadratic equation, with strictly positive determinant

```python
# 1. Define the coefficients (a,b,c) of the quadratic equation
a = 2
b = -2
c = -24

# 2. Compute the discriminant delta
delta = b**2 - 4*a*c

# 3.Print delta to check it is strictly positive
print(delta)

# 4. Compute the two roots (x1, x2)
x1 = (-b + delta**0.5)/(2*a)
x2 = (-b - delta**0.5)/(2*a)

# 5. Print the roots
print(x1)
print(x2)
```

```
196
4.0
-3.0
```

The solved notebook is in the Activity Solutions subfolder!

Single line comments

- Single line comments **help the reader understand** what is happening in your code.
- On heavy programming projects, with several programmers, these are **simply essential**.

Single line comments

- Single line comments **help the reader understand** what is happening in your code.
- On heavy programming projects, with several programmers, these are **simply essential**.
- Single line comments are also great if you want to use **separators** in your code. Personally, I use

 # ----------

Single line comments

- Single line comments **help the reader understand** what is happening in your code.
- On heavy programming projects, with several programmers, these are **simply essential**.
- Single line comments are also great if you want to use **separators** in your code. Personally, I use

 # --

```
"""
Author: Matthieu DE MARI

Description: computes and prints the determinant and the roots,
of a given quadratic equation ax^2 + bx + c = 0,
with strictly positive determinant.

Inputs: Requires the user to specify values of the parameters
a, b, and c. These should be numbers, ideally floats.

Outputs: this script calculates and prints the values of the
determinant and both roots.

Important note: if the determinant is not strictly positive,
then the roots will be incorrect.
"""


# ----------------------------------------
# 1. Define the coefficients (a,b,c) of
# the quadratic equation

a = 2
b = -2
c = -24

# ----------------------------------------
# 2. Compute the discriminant delta

delta = b**2 - 4*a*c

# ----------------------------------------
# 3.Print delta to check it is strictly positive

print(delta)

# ----------------------------------------
# 4. Compute the two roots (x1, x2)

x1 = (b + delta**0.5)/(2*a)
x2 = (b - delta**0.5)/(2*a)

# ----------------------------------------
# 5. Print the roots

print(x1)
print(x2)
```

Block comments

- **Block comments** are comments starting and ending with **triple quotation marks (“””)**.
- These are great for **headers** in your files: specifying the author name, a brief description of what the script does, etc.

```
"""
Author: Matthieu DE MARI

Description: computes and prints the determinant and the roots,
of a given quadratic equation ax^2 + bx + c = 0,
with strictly positive determinant.

Inputs: Requires the user to specify values of the parameters
a, b, and c. These should be numbers, ideally floats.

Outputs: this script calculates and prints the values of the
determinant and both roots.

Important note: if the determinant is not strictly positive,
then the roots will be incorrect.
"""
```

Block comments

- **Block comments** are comments starting and ending with **triple quotation marks (“””)**.
- These are great for **headers** in your files: specifying the author name, a brief description of what the script does, etc.

- All comments are **ignored** when executing the code, they only serve for description.

```
"""
Author: Matthieu DE MARI

Description: computes and prints the determinant and the roots,
of a given quadratic equation ax^2 + bx + c = 0,
with strictly positive determinant.

Inputs: Requires the user to specify values of the parameters
a, b, and c. These should be numbers, ideally floats.

Outputs: this script calculates and prints the values of the
determinant and both roots.

Important note: if the determinant is not strictly positive,
then the roots will be incorrect.
"""


# ----------------------------------------
# 1. Define the coefficients (a,b,c) of
# the quadratic equation

a = 2
b = -2
c = -24

# ----------------------------------------
# 2. Compute the discriminant delta

delta = b**2 - 4*a*c

# ----------------------------------------
# 3.Print delta to check it is strictly positive

print(delta)

# ----------------------------------------
# 4. Compute the two roots (x1, x2)

x1 = (b + delta**0.5)/(2*a)
x2 = (b - delta**0.5)/(2*a)

# ----------------------------------------
# 5. Print the roots

print(x1)
print(x2)
```

Matt's Great advice #3

Matt's Great Advice #3: comment your code.

It is always a good idea to **comment** your code, to let the reader know what your script does.

Either use **#** for **single line comments** or triple quotations (**“””**) for **block comments**.
You can even use **separators** (**# ------**), for readability, if needed.

Again, your future self, and colleagues, will thank you, when they work on your code later on.

About the **print()** function

- You can print multiple variables at once, in a single print(), by separating the variables with commas.

```
1  # Some values
2  a = 10
3  b = 8
4  print(a, b)
```

```
10 8
```

About the **print()** function

- You can print multiple variables at once, in a single print(), by separating the variables with commas.

```
1  # Some values
2  a = 10
3  b = 8
4  print(a, b)
```

```
10 8
```

- You can even mix several types of variables.
- It is actually a good idea, to let the user know what you are printing!

```
1  # Some more values
2  a = 10
3  b = 8
4  c = 4
5  message = "The values of (a,b,c) are:"
6  print(message, a, b, c)
```

```
The values of (a,b,c) are: 10 8 4
```

About the **print()**, and the **format()** function

- The **format()** function can be used to **insert** the value of a variable in another string variable.

```
1  a = 10
2  b = 8
3  c = 4
4  message = "The values of (a,b,c) are: {}, {}, and {}".format(a, b, c)
5  print(message)
```

```
The values of (a,b,c) are: 10, 8, and 4
```

About the **print()**, and the **format()** function

- The **format()** function can be used to **insert** the value of a variable in another string variable.
- It requires **some placeholders {}** in the string, and **some variables** passed to the **format()** method. Placeholders are simply replaced with the values of the variables passed to the format() method.

```
1  a = 10
2  b = 8
3  c = 4
4  message = "The values of (a,b,c) are: {}, {}, and {}".format(a, b, c)
5  print(message)
```

```
The values of (a,b,c) are: 10, 8, and 4
```

About the **print()**, and the **format()** function

- The **format()** function can be used to **insert** the value of a variable in another string variable.
- It requires **some placeholders {}** in the string, and **some variables** passed to the **format()** method. Placeholders are simply replaced with the values of the variables passed to the format() method.
- Note: **format()** is a **special kind of function** called a **method**. Notice how it is applied to a string variable using the **dot operator**.

```
1  a = 10
2  b = 8
3  c = 4
4  message = "The values of (a,b,c) are: {}, {}, and {}".format(a, b, c)
5  print(message)
```

```
The values of (a,b,c) are: 10, 8, and 4
```

Getting values from a user with **input()**

- You can also have Python ask the user for values explicitly.
- This is done with the **input()** function.

```
1 message = "Please enter a number and press enter: "
2 user_value = input(message)
3 second_message = "The number you entered is {}.".format(user_value)
4 print(second_message)
```

```
Please enter a number and press enter: 5
```

```
1 message = "Please enter a number and press enter: "
2 user_value = input(message)
3 second_message = "The number you entered is {}.".format(user_value)
4 print(second_message)
```

```
Please enter a number and press enter: 5
The number you entered is 5.
```

Getting values from a user with **input()**

- You can also have Python ask the user for values explicitly.
- This is done with the **input()** function.

Notice how it

- first **displays** the string variable message and submission box,
- and later **assigns** the value typed by the user in the variable on the left-hand side of the equal sign used for input().

```
1 message = "Please enter a number and press enter: "
2 user_value = input(message)
3 second_message = "The number you entered is {}.".format(user_value)
4 print(second_message)
```

```
Please enter a number and press enter: 5
```

```
1 message = "Please enter a number and press enter: "
2 user_value = input(message)
3 second_message = "The number you entered is {}.".format(user_value)
4 print(second_message)
```

```
Please enter a number and press enter: 5
The number you entered is 5.
```

Getting values from a user with **input()**

- Be careful when getting values with **input()**!
- In this program,
  1. I asked the user to enter a number,
  2. Doubled the value of said number,
  3. And later displayed both the original number and its doubled value.

```
1 message = "Please enter a number and press enter: "
2 user_value = input(message)
3 doubled_value = 2*user_value
4 second_message = "The number you entered is {}. Its doubled value is {}.".format(user_value, doubled_value)
5 print(second_message)
```

```
Please enter a number and press enter: 5
```

Getting values from a user with **input()**

- Be careful when getting values with **input()**!
- In this program,
  1. I asked the user to enter a number,
  2. Doubled the value of said number,
  3. And later displayed both the original number and its doubled value.
- Something strange occurs… But why?

```
1 message = "Please enter a number and press enter: "
2 user_value = input(message)
3 doubled_value = 2*user_value
4 second_message = "The number you entered is {}. Its doubled value is {}.".format(user_value, doubled_value)
5 print(second_message)
```

```
Please enter a number and press enter: 5
The number you entered is 5. Its doubled value is 55.
```

```
Please enter a number and press enter: 5
The number you entered is 5. Its doubled value is 55.
```

```
Its doubled value is 55.
```

```
55.
```

Has Python gone mad?

```
Please enter a number and press enter: 5
The number you entered is 5. Its doubled value is 55.
```

```
Its doubled value is 55.
```

```
55.
```

Has Python gone mad?

Getting values from a user with **input()**

- **Important:** The values retrieved from users with **input()** will always be typed as **string variables**!

Reminder! (Python basic operators)

By default, Python comes with a few **basic math operators, for number (int/float) variables**.

- **+:** addition
- **-:** subtraction
- ***:** multiplication
- **/:** division
- **//:** integer division
- **%:** remaining of the integer division (a.k.a. modulus)
- ****:** exponentiation

Watch out for the types of variables!

```
1 a = 2
2 b = "Hello World!"
3 print(a+b)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-4eaf3ef22f17> in <module>
      1 a = 2
      2 b = "Hello World!"
----> 3 print(a+b)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Getting values from a user with **input()**

- **Important:** The values retrieved from users with **input()** will always be typed as **string variables**!

- **Fun fact:** multiplying a string "5" with an integer number n simply **repeats** the string n times!
- Hence, **"5"*2** is indeed **"55".**

Getting values from a user with **input()**

- **Important:** The values retrieved from users with **input()** will always be typed as **string variables**!

- **Fun fact:** multiplying a string "5" with an integer number n simply **repeats** the string n times!
- Hence, **"5"*2** is indeed **"55".**

- **We need to convert our string variable to a float number variable before doing any maths!**

Getting values from a user with **input()**

```
1 print(type(user_value))
```

```
<class 'str'>
```

```
1 a = '5'
2 b = 2
3 c = a*b
4 print(c)
5 print(type(c))
```

```
55
<class 'str'>
```

```
1 message = "Please enter a number and press enter: "
2 user_value = input(message)
3 user_value_as_float = float(user_value)
4 doubled_value = 2*user_value_as_float
5 second_message = "The number you entered is {}. Its doubled value is {}.".format(user_value_as_float, doubled_value)
6 print(second_message)
```

```
Please enter a number and press enter: 5
The number you entered is 5.0. Its doubled value is 10.0.
```

Matt's Great advice #4

Matt's Great Advice #4: watch out for types when using basic math operators!

The behavior of a basic math **operator** (+, -, *, etc.) might **vary**, depending on the **types of the variables** involved in the calculation.

This is called **operator overloading,** and is a perfectly normal feature of Python programming.

For instance, **multiplying a string and a number variables, is not the same as multiplying two number variables!**

Activity 2: Ask users about personal details

- **Task:** Write a Python script that explicitly asks the user for its name and its age.
- Later on, the scripts should display a message, reading

Your name is: _____, and your age is ____.

- You can use the second notebook

Activity 2 - Asking users for personal details.ipynb

Activity 2: Ask users about personal details

```
1  # 1. Ask user for its name
2  user_name = input("What is your name? ")
3
4  # 2. Ask user for its age
5  user_age = input("What is your age? ")
6
7  # 3. Create and format message to be displayed
8  message = "Your name is {}, and your age is {}.".format(user_name, user_age)
9  mean_message = "Man, you're really old."
10
11 # 4. Print the message!
12 print(message)
13 print(mean_message)
```

```
What is your name? Matthieu
What is your age? 31
Your name is Matthieu, and your age is 31.
Man, you're really old.
```

Conclusion

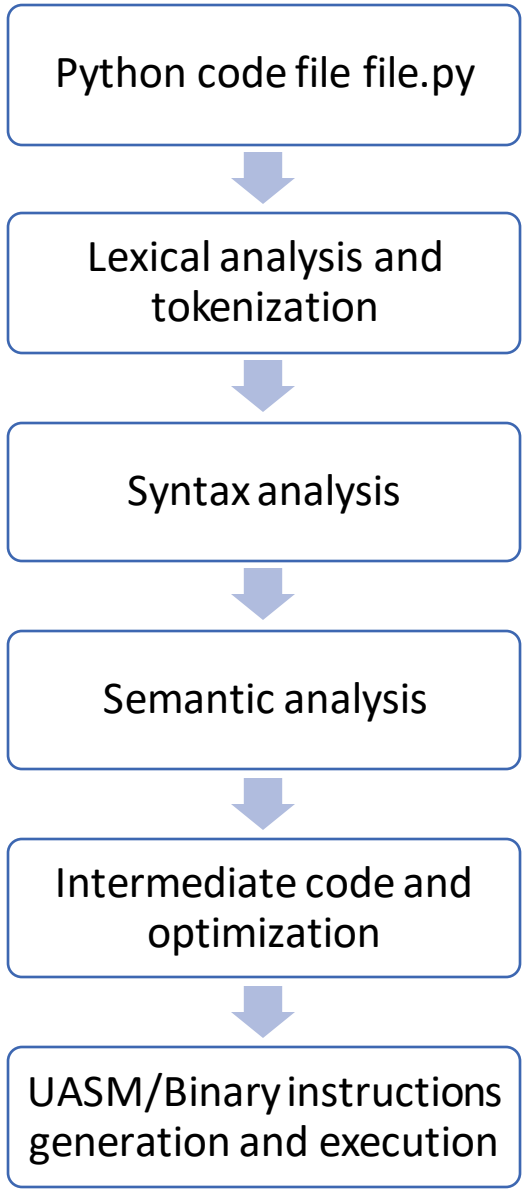
- **Variables:** Naming conventions, types, IDs, assignments, conversions
- **Basic math operators in Python:** +, -, *, /, //, %, **
- **Combining operators and assignments**
- **Commenting:** in-line, block, docstring and header comments
- **Printing and getting:** displaying with print(), getting with input()

- **Bonus (out-of-class , if time allows!):** what happens backstage when I run my code?

Decomposing the code compilation and execution procedure

From the moment you submit your code and the moment it executes, 5 steps happen

1. Lexical analysis and tokenization
2. Syntax analysis
3. Semantic analysis
4. Intermediate code and optimization
5. UASM/binary instructions generation and execution



1. Lexical analysis and tokenization

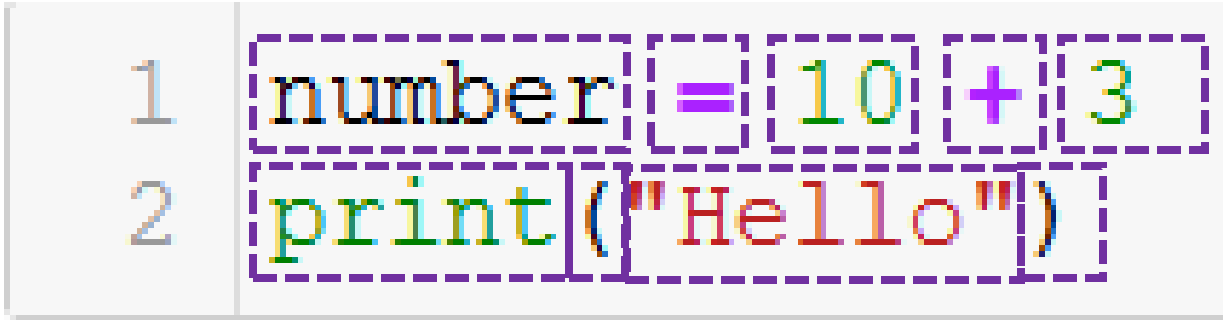
This first step

```
1  number = 10 + 3
2  print("Hello")
```

1. Lexical analysis and tokenization

This first step

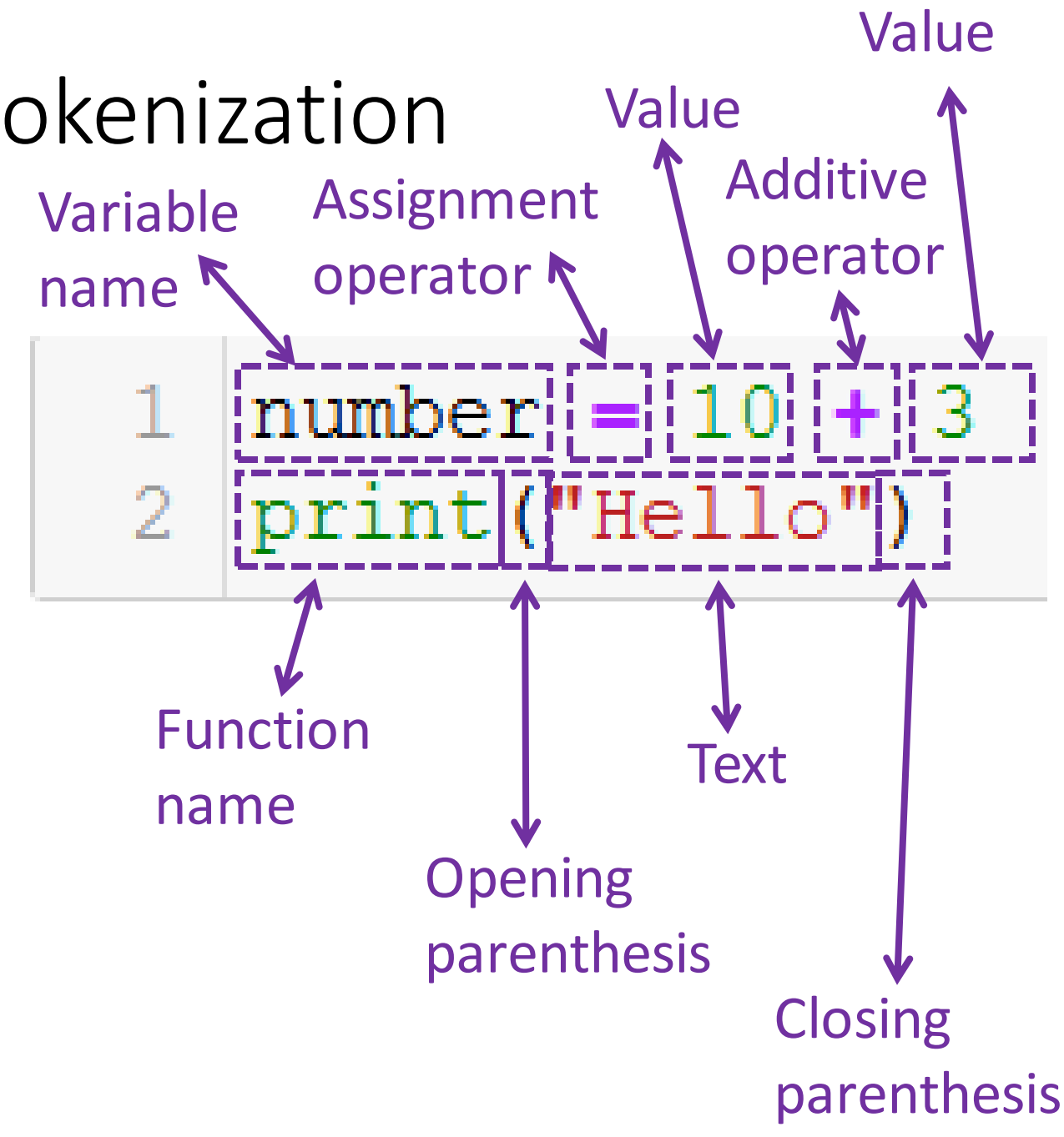
- **Breaks down the code in small pieces**, i.e. “words”
  (e.g. number, =, 10, +, 3, print, (, “Hello”,),…)



1. Lexical analysis and tokenization

This first step

- **Breaks down the code in small pieces**, i.e. “words” (e.g. number, =, 10, +, 3, print, (, “Hello”,),…)
- And **attempts to identify what each piece/”word” represents**, a.k.a. **tokenization** (e.g. number is a variable name, = is the assignment operator, 10 is a numerical value, + is the additive operator, etc.)



2. Syntax analysis

```
1  number = 10 + 3
2  print("Hello")
```

This second step

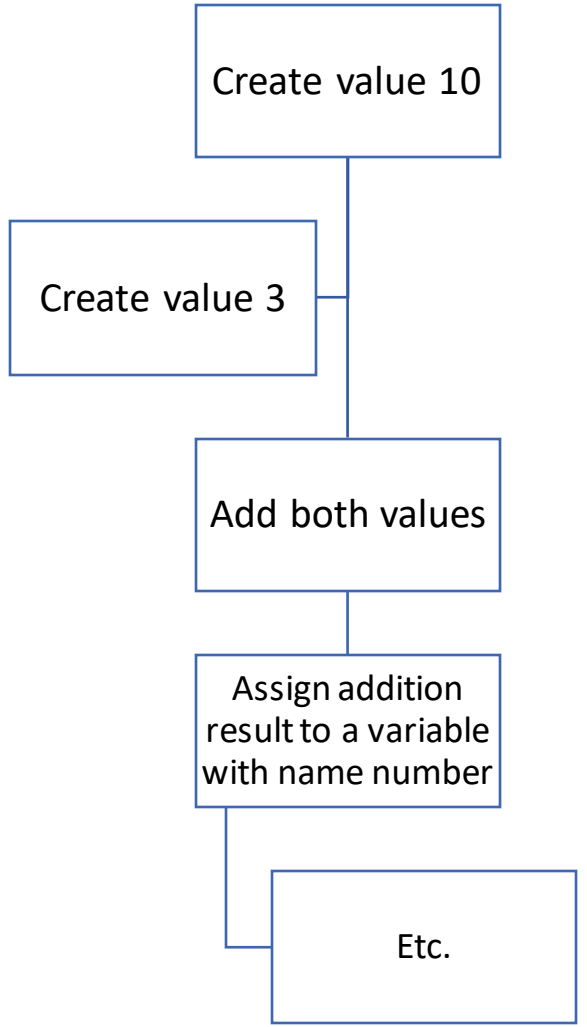
- Checks for **code structure, and language accuracy**.
  = Does it follow the expected language structure?
  (e.g. there should be a valid variable name before an assignment operator and a valid value after it, a function expects parameters between parentheses immediately after, etc.)

2. Syntax analysis

```
1  number = 10 + 3
2  print("Hello")
```

Then,

- If it passes this first step, it starts to construct a **tree structure for the operations** to be conducted.
- It defines the **successive operations** to be taken for the given code.



3. Semantic analysis

This third step checks for **semantics** (= logical meaning of the code). For instance,

```
1  number = 10 + 3
2  print("Hello")
```

3. Semantic analysis

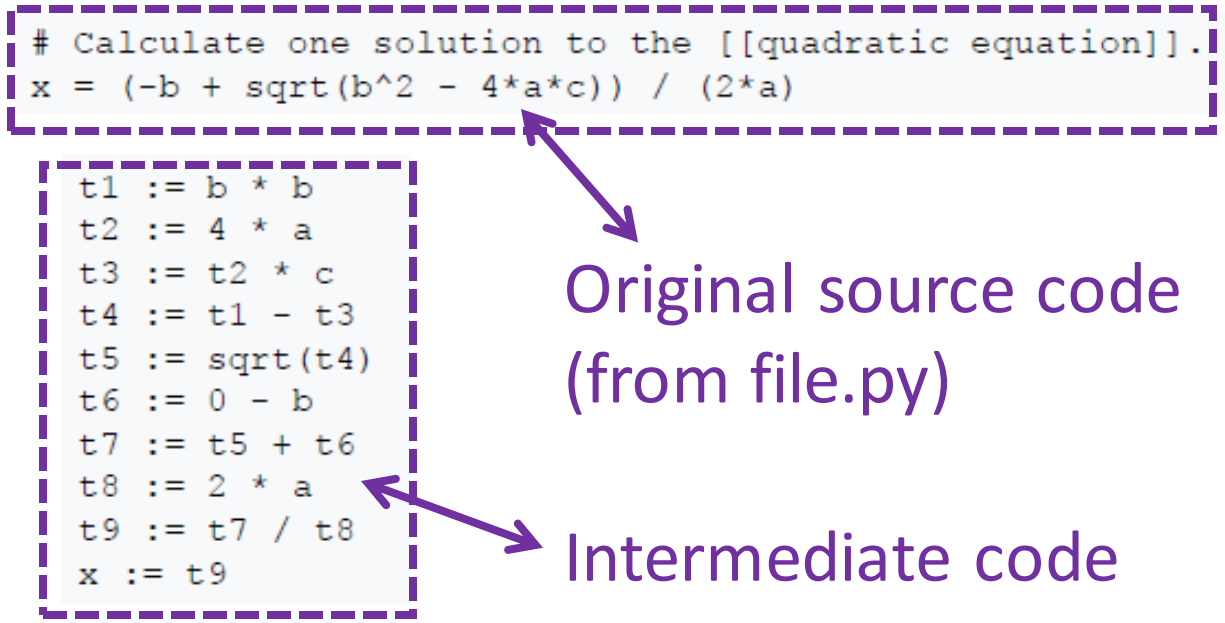
This third step checks for **semantics** (= logical meaning of the code). For instance,

- Does the function name exist?
- Are we using or printing a variable that has been previously defined?
- Is it allowed to add the values defined earlier together? (if both are numbers yes, if one is a number and the other is text, no)
- Etc.

```
1  number = 10 + 3
2  print("Hello")
```

4. Intermediate code and optimization

- **Intermediate Code:** assuming the code makes sense, convert the easy-to-read Python instructions into less readable instructions, which the computer can then execute.
- **Optimization:** Memory allocation for variables, CPU processing resource allocation and distribution.
- (Not covered, too advanced)



5. Code generation and execution

Assembly code: converts the intermediate code into binary code (0/1), which can be executed by the CPU of your computer!

(This is cumbersome to explain, but will make sense during the Computational Structures course on Term 4!)

→ **Finally, execute the assembly code and see the results!**

Compiled vs. Interpreted languages

Compiled language (e.g. C/C++)

- Scans the whole file and translates it into assembly code all at once.
- Overall, faster to compile and execute.
- Will not execute anything if there is an error somewhere, making it hard to debug.

Interpreted language (e.g. Python)

- Scans the source file, one operation at a time.
- Slower compilation and execution.
- If an error exists, it will execute the lines of code before the one that triggered the error.
- Much easier to debug, and hence more **beginner-friendly**.