IN1900 Wednesday 24/8: formulas and variables (Chapter 2)

Joakim Sundnes, Simula Research Laboratory and University of Oslo, Dept. of Informatics

Date: **Aug 24, 2022**

What will you learn in IN1900?

- General computer programming:
  - Thinking like a programmer
  - Translating mathematics to code
  - Generic concepts common to all languages
  - Debugging, testing etc.
- Python (syntax)
- Tools for programming (editor, terminal window)

Plan for august 24

- Slides/lecture: More new topics from Chapter 2 of "An introduction to..." by J. Sundnes
- "Live programming" of exercise 1.1 and 1.3 from "A primer on..." by H.P. Langtangen.

Key topics for august 24

- How to write and run a program
- *Variables* and types
- Statements
- Assignment
- Syntax and comments
- Importing modules
- Formatting output

Chapter 2 is about evaluating formulas

Why?

- Everybody understands the problem
- Many fundamental concepts are introduced
  - variables
  - arithmetic expressions
  - objects
  - printing text and numbers

Example 1: evaluate a formula

Height of a ball in vertical motion:

$$y(t) = v_0 t - \frac{1}{2} g t^2$$

where

- y is the height (position) as function of time t
- v_0 is the initial velocity at $t = 0$
- g is the acceleration of gravity

Task: Given $v_0 = 5$, $g = 9.81$ and $t = 0.6$, compute y and print it to the screen.

How to write and run the program

- A program is plain text, written in a *plain text editor*
- Use Atom, Gedit, Emacs, Vim or Spyder (*not* MS Word!)

Step 1. Write the program in a text editor, here the single line

```
print(5*0.6 - 0.5*9.81*0.6**2)
```

Step 2. Save the program to a file (say) `ball.py` . (`.py` denotes Python.)

Step 3. Move to a *terminal window* and go to the folder containing the program file.

Step 4. Run the program:

```
Terminal> python ball.py
```

The program prints out `1.2342` in the terminal window.

Alternative ways of programming Python

- *The interactive Python shell*. We have already seen this. It allows us to type Python code and test the result directly. Very useful for testing Python functionality, but not suitable for actual programming.
- *Jupyter notebooks*. This is a special document combining text with actual code. These slides are written as a a Jupyter notebook (often called an iPython notebook). The windows containing code in a notebook are run just as code you put in a file and run in the terminal window, and output from the code appears just below the window. More about this later.

Our standard way of programming is still to wite a `.py` file in an editor!

Arithmetic expressions are evaluated as you have learned in mathematics

- Example: $\frac{5}{9} + 2a^4/2$, in Python written as `5/9 + 2*a**4/2`
- Same rules as in mathematics: proceed term by term (additions/subtractions) from the left, compute powers first, then multiplication and division, in each term
- Use parentheses to override these default rules - or use parentheses to explicitly tell how the rules work: `(5/9) + (2*(a**4))/2`
- *Warning:* Missing or unmatched parentheses is one of the most common bugs in Python programs!

Store numbers in variables to make a program more readable

Our example program looked like

In [1]:
```
print(5*0.6 - 0.5*9.81*0.6**2)
```

```
1.2342
```

But from mathematics you are used to variables, e.g.,

$$v_0 = 5, \quad g = 9.81, \quad t = 0.6, \quad y = v_0 t - \frac{1}{2} g t^2$$

We can use variables in a program too, and this makes the last program easier to read and understand:

In [2]:
```
v0 = 5
g = 9.81
t = 0.6
y = v0*t - 0.5*g*t**2
print(y)
```

```
1.2342
```

This program spans several lines of text and use variables, otherwise the program performs the same calculations and gives the same output as the previous program

Defining variables in Python

- A variable is a named entity for an item of data in our program
- Variables can have different *types*, i.e. integer, float (decimal number), text string, etc.
- Technically, a variable is a name for a location in the computers memory, where the data is stored

In Python, variables are defined simply by writing their name and giving a value:

In [3]:
```
v0 = 5
g = 9.81
```

- The *type* of the variable is determined automatically by Python, based on the right hand side.

There is great flexibility in choosing variable names

- In mathematics we usually use one letter for a variable
- The name of a variable in a program can contain the letters a-z, A-Z, underscore `_` and the digits 0-9, but cannot start with a digit
- Variable names are case-sensitive (e.g., `a` is different from `A`)

In [4]:
```
initial_velocity = 5
accel_of_gravity = 9.81
TIME = 0.6
VerticalPositionOfBall = initial_velocity*TIME - \
                         0.5*accel_of_gravity*TIME**2
print(VerticalPositionOfBall)
```

```
1.2342
```

Some words are reserved in Python

Certain words have a special meaning in Python and cannot be used as variable names. These are `and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `exec`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `not`, `or`, `pass`, `print`, `raise`, `return`, `try`, `with`, `while`, and `yield`.

A program consists of statements

In [5]:
```
a = 1      # 1st statement (assignment statement)
b = 2      # 2nd statement (assignment statement)
c = a + b  # 3rd statement (assignment statement)
print(c)    # 4th statement (print statement)
```

```
3
```

Normal rule: one statement per line, but multiple statements per line is possible with a semicolon in between the statements:

In [6]:
```
a = 1;  b = 2;  c = a + b;  print(c)
```

```
3
```

Assignment statements evaluate right-hand side and assign the result to the variable on the left-hand side

In [7]:
```
myvar = 10
myvar = 3*myvar    # = 30
myvar
```

Out[7]:
```
30
```

Example 2: a formula for temperature conversion

Given $C = 21$ as a temperature in Celsius degrees, compute the corresponding Fahrenheit degrees F:

$$F = \frac{9}{5}C + 32$$

The Python program

In [8]:
```
C = 21
F = (9/5)*C + 32
print(F)
```

```
69.80000000000001
```

WARNING: Python 2 gives a different answer!

```
Terminal> python2 c2f_v1.py
53
```

Many programming languages give the same error; Java, C, C++, ...

The error is caused by (unintended) integer division

- 9/5 is not 1.8 but 1 in many computer languages (!)
- If a and b are integers, a/b implies integer division: the largest integer c such that $cb \leq a$
- Examples: $1/5 = 0$, $2/5 = 0$, $7/5 = 1$, $12/5 = 2$

- In mathematics, 9/5 is a real number (1.8) - this is called float division in Python and is the division we want

- Python 2 and many other languages will do integer division if both operands are integers
- One of the operands (a or b) in a/b must be a real number ("float") to get float division
- A float in Python has a dot (or decimals): `9.0` or `9.` is float
- No dot implies integer: `9` is an integer
- `9.0/5` yields `1.8`, `9/5.` yields `1.8`, `9/5` yields `1`

Corrected version (works in Python 2 and 3):

In [9]:
```
C = 21
F = (9.0/5)*C + 32
```

Good habit to use decimal numbers to define floats, although it is not necessary in Python 3. This will reduce the likelihood of errors if the code is later ported to another language.

Variables refer to objects. Objects have types.

Variables refer to objects. We can check the type of a variable with the function `type`:

In [10]:
```
a = 5       # a refers to an integer (int) object
b = 9       # b refers to an integer (int) object
c = 9.0    # c refers to a real number (float) object
d = b*a     # d refers to an int*int => int object
e = b/a     # e refers to int/int => float object
print(d, e)
print(type(d),type(e))
```

```
45 1.8
<class 'int'> <class 'float'>
```

We can convert between object types:

In [11]:
```
a = 3              # a is int
b = float(a)       # b is float 3.0
c = 3.9            # c is float
d = int(c)         # d is int 3
d = round(c)       # d is float 4.0
d = int(round(c))  # d is int 4
d = str(c)         # d is str '3.9'
e = '-4.2'         # e is str
f = float(e)       # f is float -4.2
```

Question for discussion

What is happening in these Python lines?

In [12]:
```
a = '10'
b = 10
```

```
print(a*10)
print(b*10)
```

```
10101010101010101010
100
```

Not all variables are numbers

We have already used strings:

In [13]:
```
a = 3                  # a is int
c = 3.9                # c is float
h = 'Hello!'           # h is string (str)
print(type(a))         # Output: <class 'int'>
print(type(c))         # Output: <class 'float'>
print(type(h))         # Output: <class 'str'>
print(type('IN1900')) # Output: <class 'str'>
```

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

Many other types, which we will see later.

Syntax is the exact specification of instructions to the computer

Programs must have correct syntax, i.e., correct use of the computer language grammar rules, and no misprints!

This is a program with two syntax errors:

In [14]:
```
myvar = 5.2
prinnt(Myvar)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-14-42f6d3aefda7> in <module>
      1 myvar = 5.2
----> 2 prinnt(Myvar)

NameError: name 'prinnt' is not defined
```

Only the first encountered error is reported and the program is stopped (correct the error and continue with next error)

Blanks may or may not be important in Python programs

These statements are equivalent (blanks do not matter):

In [ ]:
```
v0=3
v0  =  3
v0=   3
v0 = 3
```

Use blanks to make the code nice and readable for humans.

Blanks at the start of a line do matter:

In [ ]:
```
v0 = 3
    g = 9.81  #invalid, gives an error message
```

In Python, such blanks are used to group blocks of code together (more about this in Ch. 3)

Comments are useful to explain how you think in programs

Program with comments:

In [ ]:
```
# program for computing the height of a ball
# in vertical motion
v0 = 5    # initial velocity
g = 9.81  # acceleration of gravity
t = 0.6   # time
y = v0*t - 0.5*g*t**2  # vertical position
print(y)
"""
Comments can also be put inside a triple quoted
string
"""
```

Note:

- Everything after `#` on a line is a comment and ignored by Python
- Comments are used to explain what the computer instructions mean, what variables mean, how the programmer reasoned when she wrote the program, etc.
- Bad comments say no more than the code: `a = 5  # set a to 5`

Example 3: What if we need a more advanced math formula?

- What if we need to compute $\sin x$, $\cos x$, $\ln x$, etc. in a program?
- Such functions are available in Python's `math` module
- In general: lots of useful functionality in Python is available in modules - but modules must be *imported* in our programs

Task: Evaluate $Q = \sin x \cos x + 4 \ln x$ for $x = 1.2$, and print the result to the screen.

Three alternative ways:

In [ ]:
```
from math import sin, cos, log
x = 1.2
Q = sin(x)*cos(x)+4*log(x)  #log is ln (base e)
print(Q)
```

In [ ]:
```
import math
x = 1.2
Q = math.sin(x)*math.cos(x)+math.log(x)
```

In [ ]:
```
from math import *
x = 1.2
Q = sin(x)*cos(x)+4*log(x)
```

Question for discussion

Why three different methods? Why not simply use this one all the time, to minimize typing?

In [ ]:
```
from math import *
```

Hint: we will often import multiple modules in a single program, sometimes even multiple modules containing mathematical functions.

Example 4: formatting of output

Output from calculations often contain text and numbers, e.g.,

```
At t=0.6 s, y is 1.23 m.
```

Task: assign values to two variables; $t = 0.6$ and $y = 1.2342$. Print the values as indicated above, with one decimal for t and two for y.

Python's f-string gives control over the output

In [ ]:
```
t = 0.6;  y = 1.2342
print(f'At t={t} s, y is {y} m.')
```

- The `f`-prefix before the string indicates that this is a special type of string variable.
- Everything inside the curly brackets is treated as Python expressions, and their result is "inserted" into the string.
- Too many decimals for y, but we will get back to this.

The contents of the curly brackets can be any legal Python expression, which can be evaluated and return some kind of value. In the example above it was just a single variable, but it could also be a complete mathematical expression:

In [ ]:
```
t = 0.6
v0 = 5
g = 9.81
print(f'At t={t} s, y is {v0*t - 0.5*g*t**2} m.')
```

Want more control over the formatting?

We can add a *format specifier* inside the curly brackets, to control the number of decimals etc:

In [ ]:
```
t = 1.234567
print(f"Default output gives t = {t}.")
print(f"We can set the precision: t = {t:.2}.")
print(f"Or control the number of decimals: t = {t:.2f}.")
print(f"Or control the space used for the output: t = {t:8.2f}.")
```

The last one is very useful for outputting multiple lines in a table-like style, to ensure that numbers are properly aligned under eachother.

Back to the original task; print on the following format with two decimals for y:

```
At t=0.6 s, y is 1.23 m.
```

In [ ]:
```
t = 0.6; y = 1.2342

print(f"At t={t} s, y is {y:.2f} m.")
```

Similar specifiers exists for integers (using `d` for *digit*):

In [ ]:
```
r = 87
print(f"Integer output specified to take up 8 chars of space: r = {r:8d}")
```

Formatting multiple lines

Multi-line strings, defined using triple quotes, also work with the f-string formatting:

In [ ]:
```
v0 = 5
g = 9.81
t = 0.6
print(f"""v0 = {v0} m/s
g = {g} m/s^2
t = {t} s
y = {v0*t - 0.5*g*t**2} m""")
```

Alternative ways to format output

There are multiple ways to format text and number output in Python:

1. The f-string method considered above is the newest method, which was introduced in Python version 3.6. This is arguably the quickest and most convenient, and we recommend sticking to this method for everyone using Python 3.6 or newer.

1. The book by Langtangen describes the so-called *printf-formatting*, or sometimes called %-formatting. The idea is the same as with the f-strings, that we create strings with "slots" where we insert variables or expression. However, the syntax is less intuitive than the f-string. The printf-formatting originates from the C language, and has the advantage that it works in multiple programming languages.

2. In Python versions from 2.7 to 3.5, the standard way of formatting text was to use a function called `format`, which is built into the string-type (`str`). Again, the idea is exactly the same as for the f-string, but the syntax is different.

Summary of Chapter 2 (part 1)

- Programs must be accurate!
- Variables are names for objects
- We have met different object types: `int`, `float`, `str`
- Choose variable names close to the mathematical symbols in the problem being solved
- Arithmetic operations in Python: term by term (+/-) from left to right, power before * and / - as in mathematics; use parenthesis when there is any doubt
- (If you use Python 2: Watch out for unintended integer division!)

Summary of Chapter 2 (part 2)

Mathematical functions like $\sin x$ and $\ln x$ must be imported from the `math` module:

In [ ]:
```
from math import sin, log
x = 5
r = sin(3*log(10*x))
print(r)
```

Use f-strings for full control of output of text and numbers!

Important terms: object, variable, type, statement, assignment.

Summarizing example: throwing a ball (problem)

We throw a ball with velocity v_0, at an angle θ with the horizontal, from the point $(x = 0, y = y_0)$. The trajectory of the ball is a parabola (we neglect air resistance):

$$y = x\tan\theta - \frac{1}{2v_0}\frac{gx^2}{\cos^2\theta} + y_0$$

- Program tasks:
  - initialize input data (v_0, g, θ, y_0)
  - import from `math`
  - compute y
- We give x, y and y_0 in m, $g = 9.81\mathrm{m/s}^2$, v_0 in km/h and θ in degrees - this requires conversion of v_0 to m/s and θ to radians

Summarizing example: throwing a ball (solution)

In [ ]:

```
g = 9.81     # m/s**2
v0 = 15      # km/h
theta = 60   # degrees
x = 0.5      # m
y0 = 1       # m

print(f"""v0    = {v0} km/h
theta = {theta} degrees
y0    = {y0} m
x     = {x} m""")

# convert v0 to m/s and theta to radians:
v0 = v0/3.6
from math import pi, tan, cos
theta = theta*pi/180

y = x*tan(theta) - 1/(2*v0)*g*x**2/((cos(theta))**2) + y0

print(f'y = {y:.2} m')
```