PYTHON FLOW CONTROL and BOOLEANS

CSC 109 - Introduction to programming in Python - Summer 2023

Table of Contents

1. Introduction

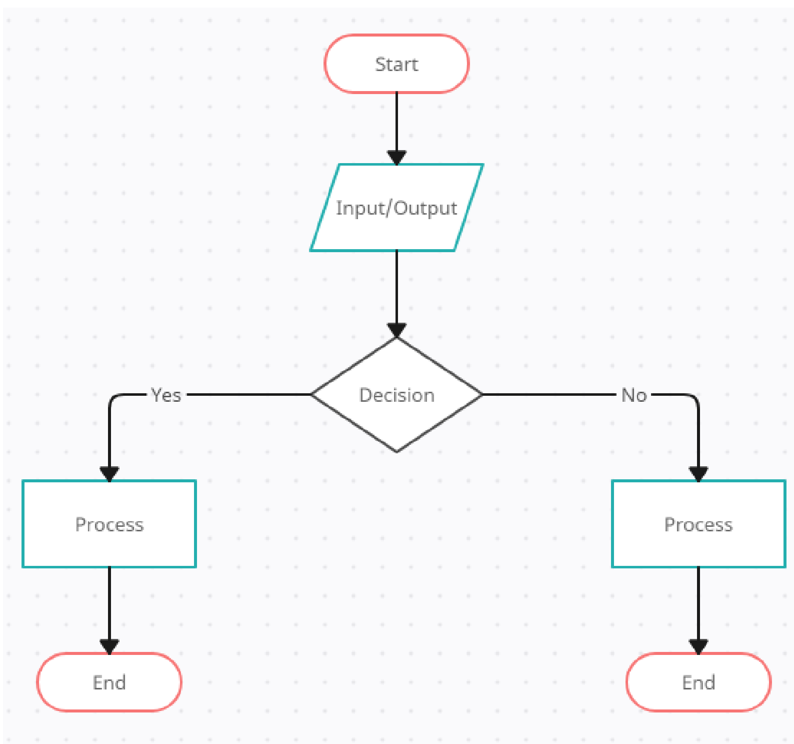


Figure 1: Flowchart template from app.creately.com

- A program is just a serious of instructions for the computer but the real power of computing comes from making decisions.
- Three common visualization methods are common:
    1. No: Flow charts (easy but not standardized, hard to read)

2. Yes: BPMN (standard, widely used for decision modeling)
3. No: UML (standard, limited to IT folk, harder to learn)

- Based on how *expressions* evaluate, the program can decide to
    1. **skip** instructions
    2. **repeat** instructions
    3. **choose** one of several instructions
- *Flow control statements* decide what to do under which conditions.

2. Visualizing flow with BPMN

- Processes can be visualized in a flowchart (or in a BPMN diagram as shown below[1]), with:

    1. events or statuses (start/end)
    2. gateways or decision points (exclusive/parallel)
    3. tasks (action like "go outside", "wait a while")
    4. sequence flow (directed lines)

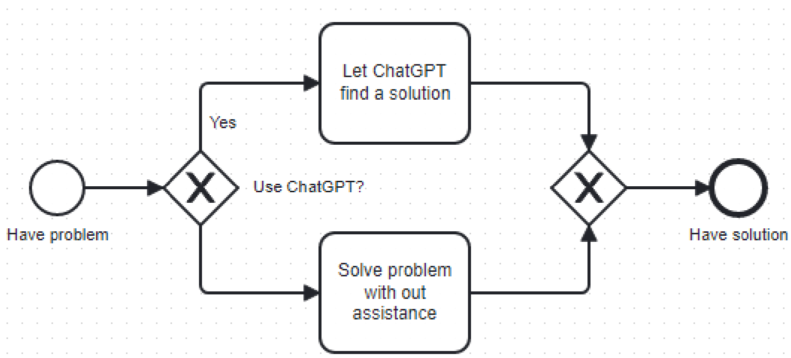


Figure 2: BPMN diagram to decide if to use ChatGPT or not

- When there are more process participants, additional elements are useful:

    1. pools and lanes (participants like user/chatbot)
    2. message events (sending/receiving stuff)
    3. message flow (directed lines between pools)

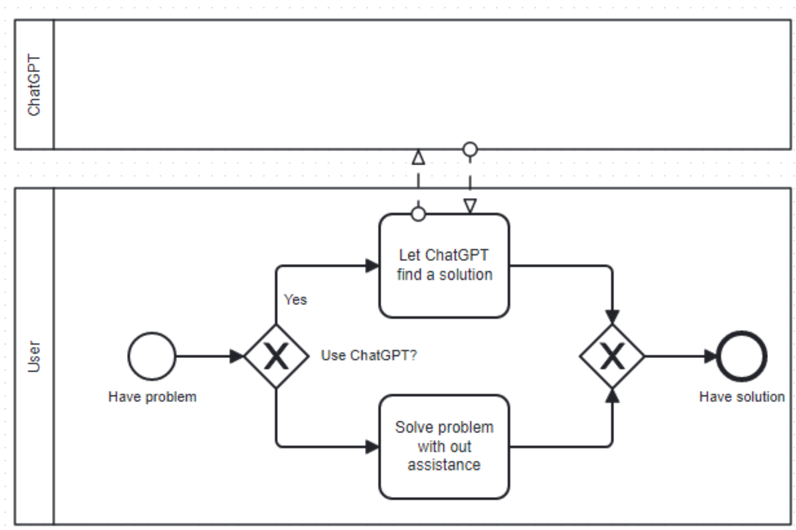
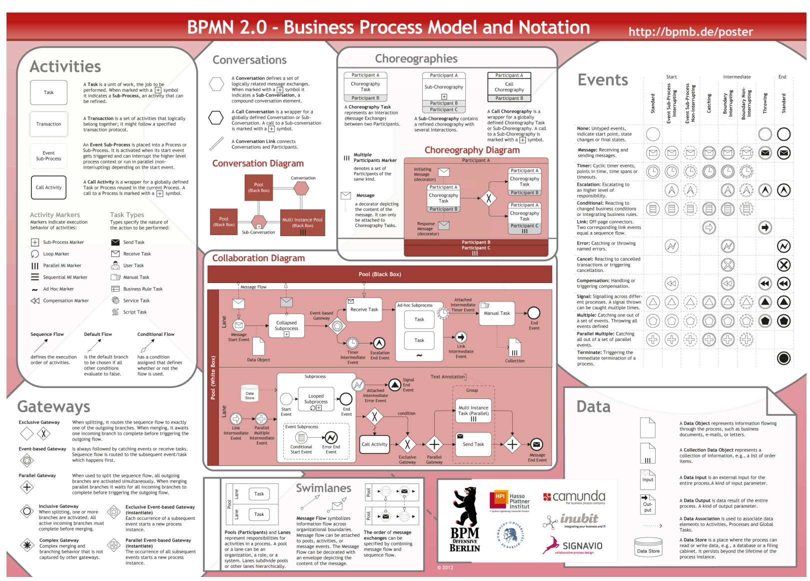


Figure 3: BPMN diagram modeling interaction with chatbot

- This type of visualization is a *modeling* activity[2]: syntactically correct BPMN diagrams can even be used to automatically create code[3].
- You can see the full language model in this cheat sheet (bpmb.de):



- What are advantages of such models? What are disadvantages?
    1. Advantages: simulate changes visually without changing code; highlight structure by eliminating detail. Identify and check process elements. Apply aesthetic criteria. Mathematical.
    2. Disadvantages: dissociated from code; specifics are missing; subjective selection of elements; static (however: see process mining).

- Python is considered to be so simple as to amount to 'pseudocode' (no syntax knowledge necessary) - really?[4]

```python
# cyclomatic_example.py
import sys

def main():
    if len(sys.argv) > 1:  # 1
        filepath = sys.argv[1]
    else:
        print("Provide a file path")
        exit(1)
    if filepath:  # 2
        with open(filepath) as fp:  # 3
            for line in fp.readlines():  # 4
                if line != "\n":  # 5
                    print(line, end="")

if __name__ == "__main__":  # Ignored.
    main()
```

- To control process flow with Python code, we need a way to check if an event has happened or not, and a way to compare events.
- Mathematically, this means that we need:
  1. Boolean values (`True` and `False`)
  2. Comparison operators (to compare anything)
  3. Boolean operators (to compare Boolean values)

3. Exercise: BPMN model

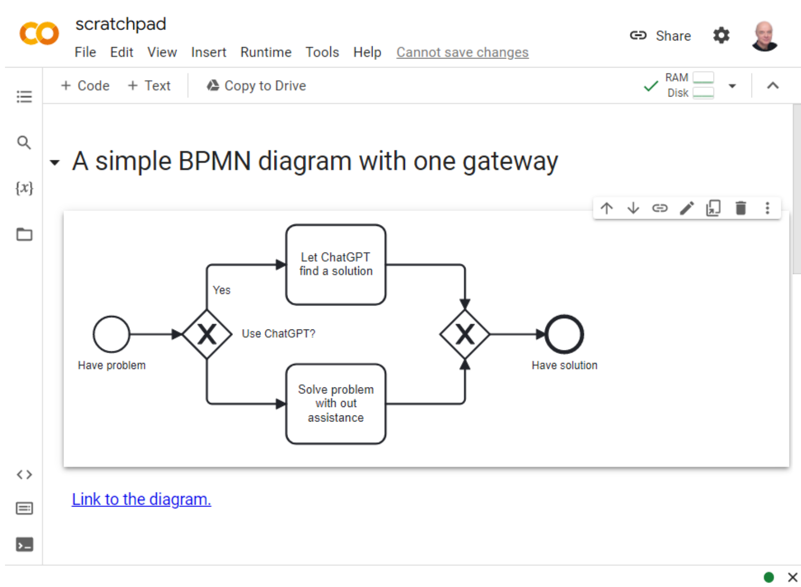
1. Model our first Python program as a BPMN model using `bpmn.io`:

   i. Says 'Hello world!'
   ii. Asks for your name
   iii. Greets you with your name
   iv. Tells you how many characters your name has
   v. Asks for your age
   vi. Tells you how old you're going to be in one year

   ```python
   print("Hello world!")
   name = input("What is your name? ")
   print("Good to meet you, " + name)
   print("Your name has ", len(name), " characters")
   age = input("What is your age? ")
   print("You're going to be " + str(int(age) + 1) + " years old")
   ```

2. Start with a pool and name it `Computer`.
3. Add suitable events and tasks connected by sequence flow.
4. Take a screenshot. It should look like this.
5. Add another pool and name it `User`.

6. Connect the two pools with suitable (message) flow.
7. Take another screenshot. It should look like this.
8. Save your diagram as `.bpmn` and as `.svg` files.
9. Add your `.svg` diagram in a titled Colab text cell:
10. Upload your diagram to a drive and link to it in the text cell.



Note: BPMN process diagram elements can be *overloaded*, i.e. given meta information, such as 'tasks accepts input' or 'task sends output' (see overloaded diagrams here and here). More about BPMN from camunda.com.

4. Boolean values

- The Boolean[5] data type only has the values `True` and `False` and must be written in exactly this way. Try this on a Python shell:

```
ham = TRUE
ham = True
ham
spam = False
spam
true
True = 2 + 2
```

- Boolean values are used in expressions and can be stored in variables of type Boolean:

```
print(type(True))
```

```
<class 'bool'>
```

5. Comparison operators

- Comparison operators are binary operators (they have a left and a right hand argument) and evaluate down to a single Boolean value: #+attr$_{\text{html}}$:width 400px

Operator	Meaning
`==`	Equal to
`!=`	Not equal to
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to

Figure 4: Comparison operators (Source: Sweigart, 2020).

- Let's try this in the shell - when you type each command, think about what the answer might be before you type ENTER:

```
42 == 42
42 == 'Hello'
42 == 41
2 != 1
42 < 100
42 >= 100
42 < 42
42 <= 42
0 == 1e-350
0 == 1e-300
```

- With variables: comparisons are expressions and evaluate to a single (Boolean) value no matter what:

```
myAge = 59  # a statement
myAge < 60  # an expression
```

- Integers and strings are never equal to one another:

```
print(42 == '42')
```

```
False
```

- How can you get `42 == '42'` to evaluate to `True`?

```python
print(str(42) == '42')
print(42 == int('42'))
```

```
True
True
```

- Float and integer values can be equal to one another:

```python
print(42.0 == 42)
```

```
True
```

- However, the `<`, `>`, `<=`, and `>=` operators only work properly with integer and floating-point values on either side:

```python
print(42.0 < 42)
print(42.0 > 42)
```

```
False
False
```

6. Boolean operators

#+attr$_{html}$:width 400px

Expression	Evaluates to . . .
`True and True`	`True`
`True and False`	`False`
`False and True`	`False`
`False and False`	`False`

Figure 5: Table with Boolean operators (Source: Sweigart, 2020).

- The `and` and `or` operators are *binary* (they take two values) like arithmetic operators, while the `non` operator is *unary*.
- Test the `and` operator and the `or` operator in a Python shell. #+attr$_{html}$:width 400px

Expression	Evaluates to . . .
`True or True`	`True`
`True or False`	`True`
`False or True`	`True`
`False or False`	`False`

Figure 6: Table with Boolean operators (Source: Sweigart, 2020).

- The `and` operator only leads to `True` if both values are `True`, while the `or` operator only leads to `False` if both values are `False`.
- The `not` operator evaluates to the opposite Boolean value: #+attr$_{html}$:width 400px

Expression	Evaluates to . . .
`not True`	`False`
`not False`	`True`

Figure 7: Table with Boolean operators (Source: Sweigart, 2020).

- In code:

```
print(not True)
print(not False)
```

```
False
True
```

- The Boolean `not`, `and`, `or` operators have the lowest precedence of all operators - what'll the output be of these expressions?

```
print(not True == False)
print(not True == False + 1)
print((not True == False) + 1)
```

```
True
False
```

```
2
```

- What will the output be of this expression?[6]

```
print(True == not True)
```

- **Exercise:** Open a Colab notebook and check if De Morgan's laws are implemented in Python: #+attr$_{html}$:width 400px

$$\neg(P \lor Q) \iff (\neg P) \land (\neg Q),$$

$$\neg(P \land Q) \iff (\neg P) \lor (\neg Q)$$

Figure 8: De Morgan's laws (Wikipedia).

- `Bonus:` in a text cell, include the logic formula in $LaTeX$ (here is a list of mathematical LaTeX symbols):
  1. `not` is $\neg$
  2. `and` is $\wedge$
  3. `or` is $\vee$
  4. `==` is $\iff$
- Remember that you can copy and paste whole text and code cells!
- Solution in Python code:

```
# NOT (P OR Q) <=> NOT(P) AND NOT(Q)
print(not(True or True) == (not True and not True))
print(not(True or False) == (not True and not False))
print(not(False or True) == (not False and not True))
print(not(False or False) == (not False and not False))
# NOT (P AND Q) <=> NOT(P) OR NOT(Q)
print(not(True and True) == (not True or not True))
print(not(True and False) == (not True or not False))
print(not(False and True) == (not False or not True))
print(not(False and False) == (not False or not False))
```

```
True
True
True
True
True
True
True
True
```

- **Bonus exercise (home):** Instead of printing `True` after each statement, show that De Morgan's laws hold, but this time:
  1. print only the number of `True` statements at the end.

2. print the final statement using string concatenation
3. print the final statement using an 'f-string'

- The *exclusive* gateway that you saw in the BPMN diagram earlier, is the result of a composite Boolean operation. It is only `True` if either of the two values are `True`, and `False` otherwise.

- This combination of Boolean operators does that ∀ Booleans p, q: #+attr$_{html}$:width 400px

$$(p \lor q) \land (\neg p \lor \neg q)$$

Figure 9: Exclusive OR operation (Wikipedia)

- You can test if this is implemented in Python as before:

```
print((True or True) and (not True or not True))  # A = B = True
print((True or False) and (not True or not False)) # A=True, B=False
print((False or True) and (not False or not True)) # A=False, B=True
print((False or False) and (not False or not False)) # A = B = False
```

```
>>> False
True
True
False
```

- Fortunately, Python has an bit-wise XOR ('exclusive `or`) operator[7]:

```
print(True ^ True)
print(True ^ False)
print(False ^ True)
print(False ^ False)
```

```
False
True
True
False
```

7. Compound logical operators

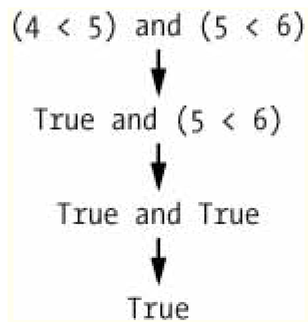
- Comparison and Boolean operators can be mixed to establish more complicated logical dependencies.

```
print((4 < 5) and (5 < 6))
print((4 < 5) and (9 < 6))
print((1 == 2) or (2 == 2))
```

```
True
False
```

```
True
```

- Here is the evaluation process of the computer:



- What will the output be? What's the order or precedence?

```
result = 5 < 10 and 2 + 2 == 4 or not (3 >= 5)
print(result)
```

```
True
```

- Order or evaluation:

```
2 + 2  # 4 (True)
5 < 10 # True
3 >= 5 # False
4 == 4 # True
not False  # True
True and True # True
True or True # True
```

- Compound logical expressions are common in database queries to filter records that satisfy several conditions for different features - here is an SQLite example:

```
-- .databases -- check database
-- CREATE TABLE people -- create table
--        (f_name TEXT, l_name TEXT,
--         century text, phy INTEGER, eng INTEGER);
-- .tables -- check tables
-- INSERT INTO people VALUES ("Albert","Einstein","19",TRUE,FALSE);
-- INSERT INTO people VALUES ("Elon","Musk","20",FALSE,TRUE);
-- INSERT INTO people VALUES ("Nikola","Tesla","19",TRUE,TRUE);
-- ---------------------------------------------------------------
.mode box
SELECT * FROM people; -- return only people born in the 19th century
-- who were both physicists and engineers:
SELECT * FROM people WHERE born=="19" AND eng==TRUE AND phy==TRUE;
```

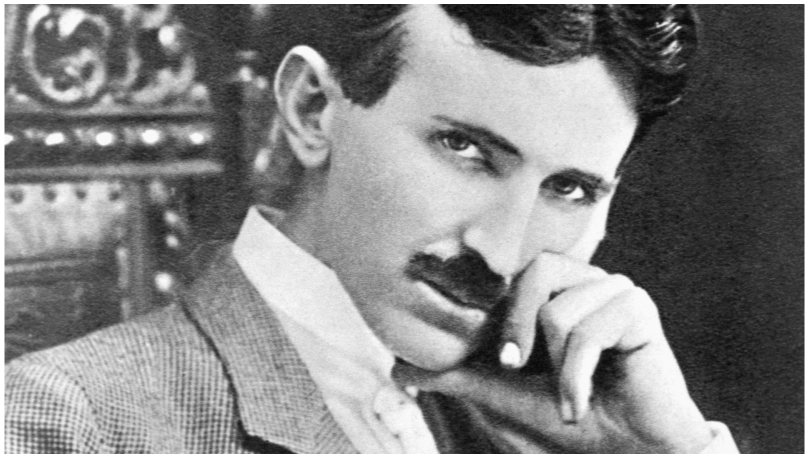


Figure 10: Nikola Tesla (1856-1943)

- For example, to test if someone's age is both greater than 20 and if he owns a cat:

```
age = 22
pet = 'cat'
print(age > 20 and pet == 'cat')
```

```
True
```

- Exercise! Let's say Joe is 20 and Jane is 24 years old, Joe has a dog, and Jane has a cat:

  1. Establish suitable variables for Joe and Jane
  2. Assign the correct values to these variables
  3. Assign ALL of these values on ONE line only

```
# Assign age and pet for Joe and Jane
age_joe, pet_joe, age_jane, pet_jane = 20, 'dog', 24, 'cat'
```

- Using these variables and their values, check:

  1. Does Jane have a dog?
  2. Is Joe younger or as old as Jane?
  3. Is Jane as old as Joe, and do they have different pets?
  4. Is Jane older than Joe, or is Jane's pet a dog?

```
# Does Jane have a dog?
print(pet_jane == 'dog')
# Is Joe younger or as old as Jane?
print(age_joe <= age_jane)
# Is Jane as old as Joe, and do they have different pets?
print((age_jane == age_joe) and (pet_jane != pet_joe))
# Is Jane older than Joe, or is Jane's pet a dog?
print((age_jane >= age_joe) or (pet_jane == 'dog'))
```

```
False
True
False
True
```

- Lastly, check if 4 is 2+2 and 2*2, and 2+2 is not 5:

```
print(2 + 2 == 4 and 2 * 2 == 4 and not 2 + 2 == 5)
```

```
True
```

8. Summary

- The Boolean data type has only two values: `True` and `False` (both beginning with capital letters).
- Comparison operators compare two values and evaluate to a Boolean value: `==`, `!=`, `<`, `>`, `<=`, `>=`
- `==` is a comparison operator, while `=` is the assignment operator for variables.
- Boolean operators (`and`, `or`, `not`) also evaluate to Boolean values.

9. Glossary

TERM/COMMAND	MEANING
`True`	Boolean non-Null, truth
`False`	Boolean Null/empty, falsehood
`==`	Comparison: equality
`!=`	Comparison: in-equality
`<`, `<=`, `>`, `>=`	Comparison: relations
`in`, `not in`	Comparison: containment
`not`, `and`, `or`	Boolean operators

10. References

- IBM (2023). BPEL process. URL: ibm.com.
- Camunda (2022). Web-based tooling for BPMN, DMN and Forms. URL: bpmn.io.

Footnotes:

[1] BPMN stands for "Business Process Model and Notation" and is a standardized, diagrammatic language especially suited to modeling business processes. Correct BPMN diagrams can be auto-translated into code using BPEL (Business Process Execution Language) - see OMG, 2010. For more information see here, and to try it see here.

[2] For a (new, short) book on modeling in Python, see Downey, Modeling and Simulation in Python (NoStarch, 2023), free online. It is also one of the textbooks for DSC 482.02 Data and Process Modeling (fall 2023).

[3] IBM has developed a language called BPEL (Business Process Execution Language) that facilitates this process (IBM, 2023).

[4] The code example (from Shaw, 2019) has 'cyclomatic complexity' of 5, i.e. there are 5 independent code paths that the Python interpreter can follow to get to the end of the application.

[5] This type name is capitalized because it is named after the mathematician George Boole (1815-1864) who found Boolean algebra, which can be used to design circuits in terms of logic gates.

[6] The `not` operator is a unary operator and requires an operand immediately after it - that's not what the computer sees here because it evaluates from left to right. Fixes:

```
True == (not True)
not True == True
```

[7] This operator does something else if fed with binary numbers.

Author: Marcus Birkenkrahe

Created: 2023-05-31 Wed 11:11