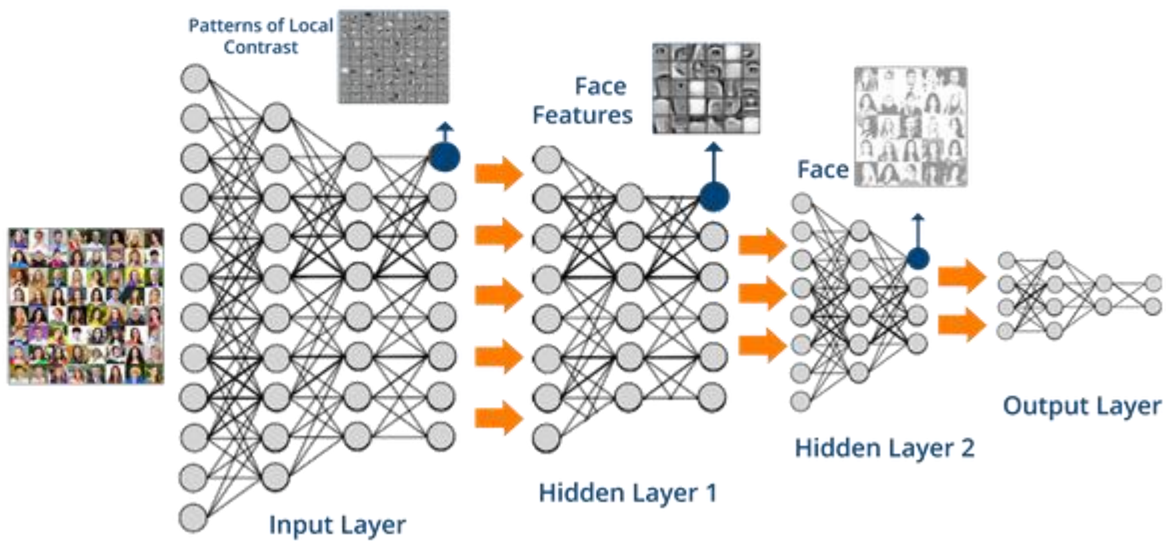
Control Structures

Selection Control

Can computers Think?



Computers are as powerful as the algorithm they run!!

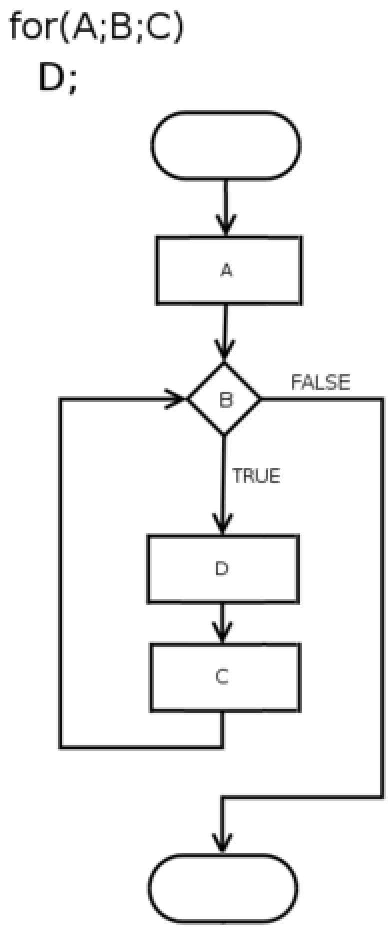
What is control flow?

- Control flow --> order that instructions are executed in a program

Escenario 1

Escenario 2

What is control flow?



A. Escoger medio de transporte

B. ¿Esta bloqueado?

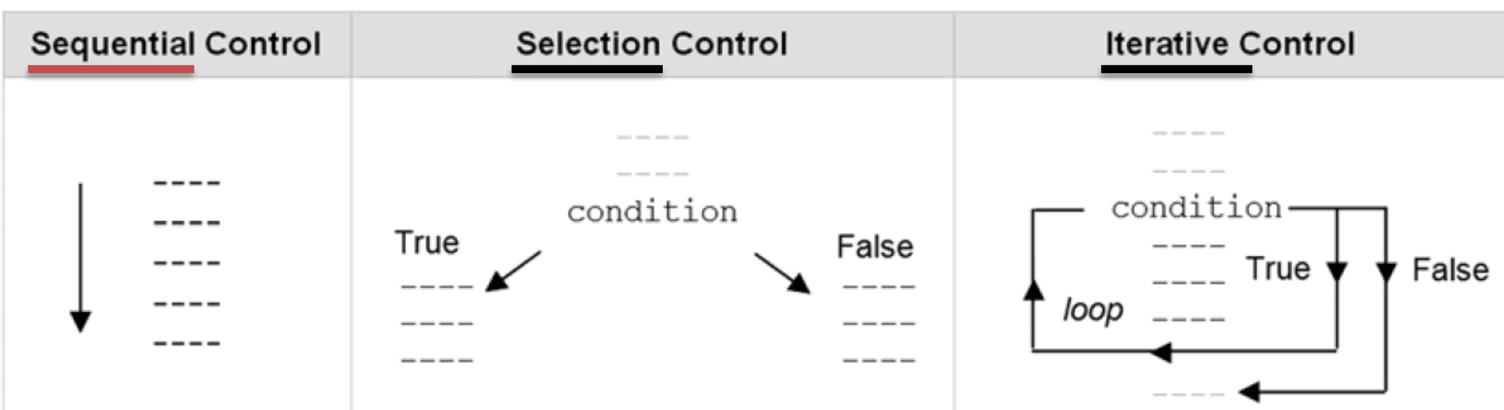
C. Escoger un medio de trasporte

D. Ir al medio de transporte

What is control flow?

- A **control statement** is a statement that determines the control flow of a set of instructions.

STATEMENT TO SPECIFY HOW CONTROL FLOW SHOULD CHANGE



Selection Control

THE MATHEMATICAL ANALYSIS

OF LOGIC,

BEING AN ESSAY TOWARDS A CALCULUS
OF DEDUCTIVE REASONING.

BY GEORGE BOOLE.

Ἐπικοινωνοῦσι δὲ πᾶσαι αἱ ἐπιστῆμαι ἀλλήλαις κατὰ τὰ κοινά. Κοινὰ δὲ λέγω, οἷς χρῶνται ὡς ἐκ τούτων ἀποδεικνύντες· ἀλλ' οὐ περὶ ὧν δεικνύουσιν, οὐδὲ ὃ δεικνύουσι.

ARISTOTLE, *Anal. Post.*, lib. I. cap. XI.

CAMBRIDGE:
MACMILLAN, BARCLAY, & MACMILLAN;
LONDON: GEORGE BELL.

1847

"Todas las ciencias se asocian con otras respecto a elementos comunes. (Y yo llamo común a todo aquello que utilizan en sus demostraciones, no a aquello que puede ser o no ser probado)"

Conditions and Boolean Expressions

- The **Boolean data type** contains two Boolean values, denoted as **True** and **False** in Python.
- A **Boolean expression** is an expression that evaluates to a Boolean value.

`num = 10`	variable `num` is assigned the value `10`
`num == 10`	variable `num` is compared to the value `10`

Relational operators

- Relational operators perform the usual comparison operations

Relational Operators	Example	Result
`==` equal	`10 == 10`	`True`
`!=` not equal	`10 != 10`	`False`
`<` less than	`10 < 20`	`True`
`>` greater than	`'Alan' > 'Brenda'`	`False`
`<=` less than or equal to	`10 <= 10`	`True`
`>=` greater than or equal to	`'A' >= 'D'`	`False`

LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> 10 == 20
???
>>> 10 != 20
???
>>> 10 <= 20
???
```

```
>>> '2' < '9'
???
>>> '12' < '9'
???
>>> '12' > '9'
???
```

```
>>> 'Hello' == "Hello"
???
>>> 'Hello' < 'Zebra'
???
>>> 'hello' < 'ZEBRA'
???
```

Lexicographical order

Lexicographic Order

An ordering for the Cartesian product $\times$ of any two sets A and B with order relations $<A$ and $<B$, respectively, such that if (a_1, b_1) and (a_2, b_2) both belong to $A \times B$, then $(a_1, b_1) < (a_2, b_2)$ iff either

1. $a_1 < A\, a_2$, or

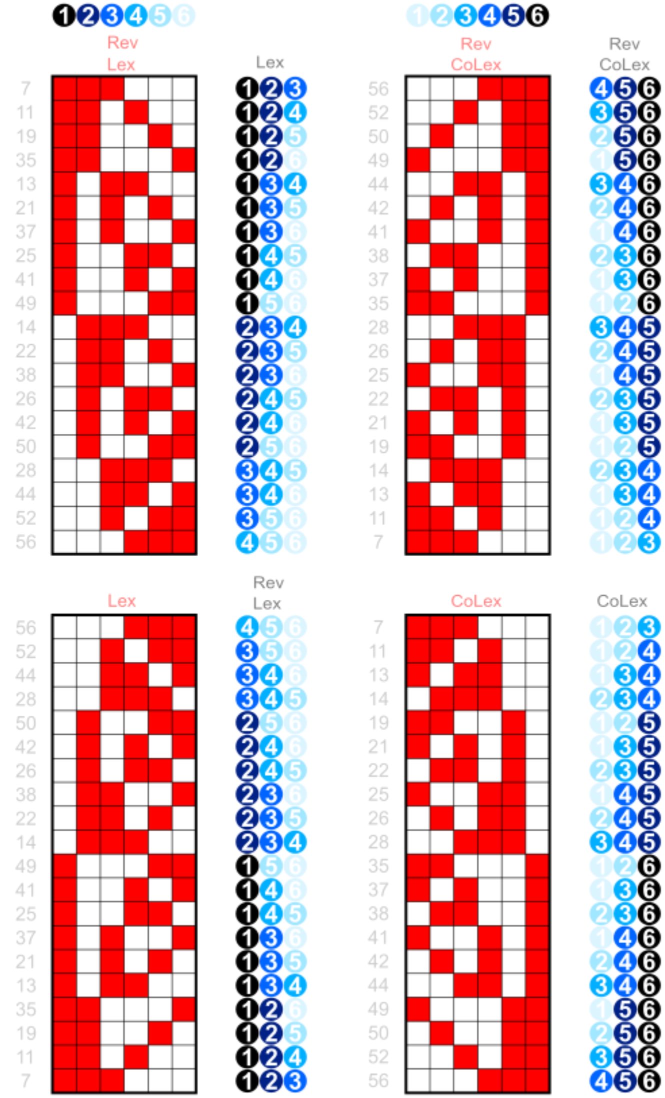
2. $a_1 = a_2$ and $b_1 < B\, b_2$.

The lexicographic order can be readily extended to cartesian products of arbitrary length by recursively applying this definition, i.e., by observing that $A \times B \times C = A \times (B \times C)$.

When applied to permutations, lexicographic order is increasing numerical order (or equivalently, alphabetic order for lists of symbols; Skiena 1990, p. 4). For example, the permutations of $\{1, 2, 3\}$ in lexicographic order are 123, 132, 213, 231, 312, and 321.

When applied to subsets, two subsets are ordered by their smallest elements (Skiena 1990, p. 44). For example, the subsets of $\{1, 2, 3\}$ in lexicographic order are $\{\}$, $\{1\}$, $\{1, 2\}$, $\{1, 2, 3\}$, $\{1, 3\}$, $\{2\}$, $\{2, 3\}$, $\{3\}$.

Lexicographic order is sometimes called dictionary order.



Finite subsets

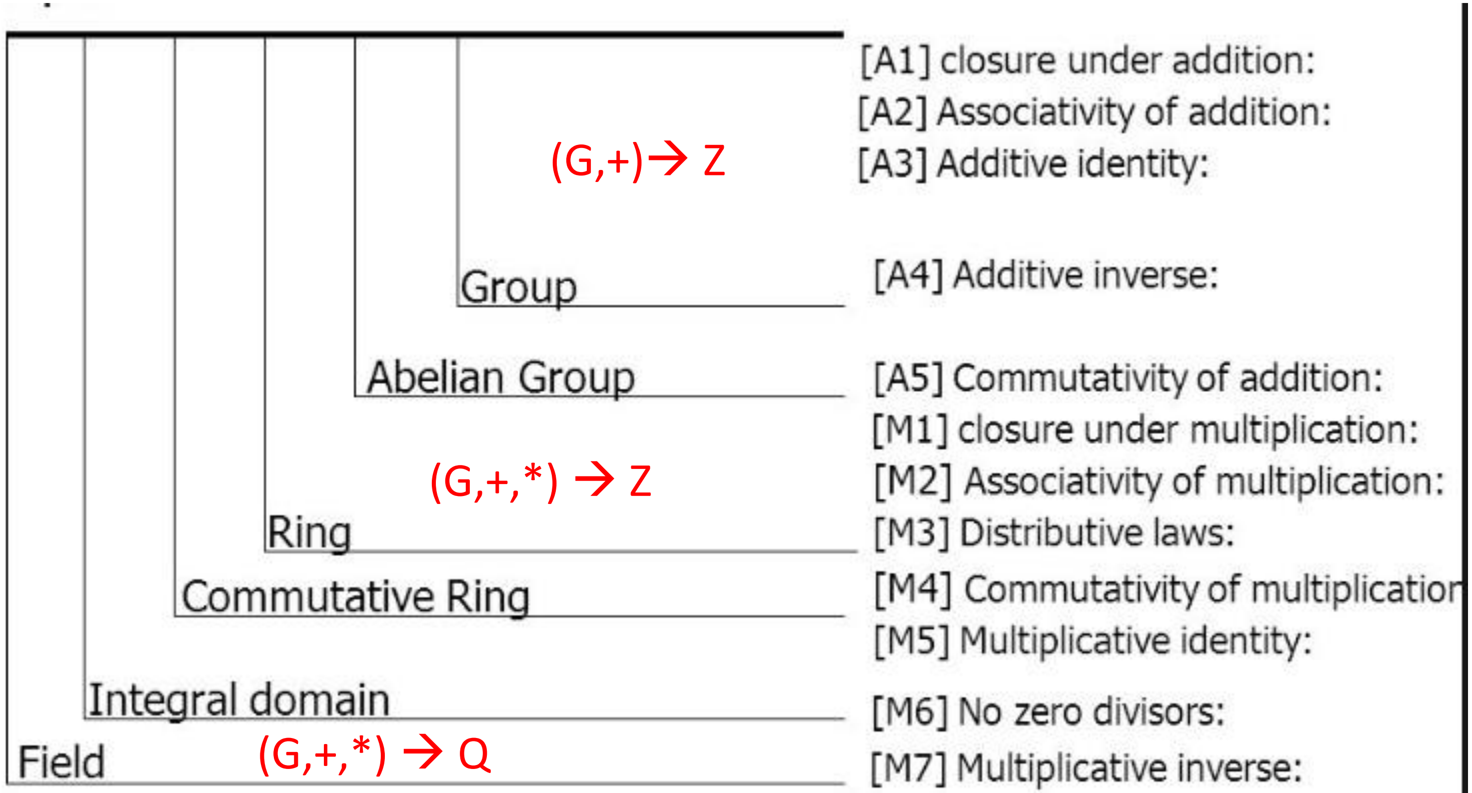
In combinatorics, one has often to enumerate, and therefore to order the finite subsets of a given set S. For this, one usually chooses an order on S. Then, sorting a subset of S is equivalent to convert it into an increasing sequence. The lexicographic order on the resulting sequences induces thus an order on the subsets, which is also called the *lexicographical order*.

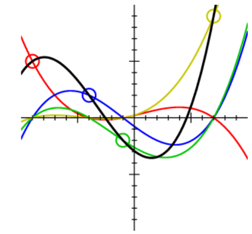
In this context, one generally prefer to sort first the subsets by cardinality, such as in the shortlex order. Therefore, in the following, we will consider only orders on subsets of fixed cardinal.

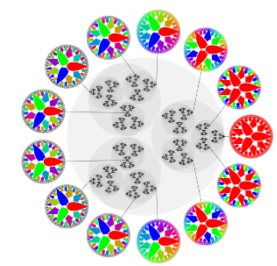
For example, using the natural order of the integers, the lexicographical ordering on the subsets of three elements of $S = \{1, 2, 3, 4, 5, 6\}$ is

$$123 < 124 < 125 < 126 < 134 < 135 < 136 < 145 < 146 < 156 <$$
$$234 < 235 < 236 < 245 < 246 < 256 < 345 < 346 < 356 < 456.$$

For ordering finite subsets of a given cardinality of the natural numbers, the *colexicographical* order (see below) is often more convenient, because all initial segments are finite, and thus the colexicographical order defines an order isomorphism between the natural numbers and the set of sets of n natural numbers. This is not the case for the lexicographical order, as, with the lexicographical order, we have, for example, $12n < 134$ for every $n > 2$.







Applications of abstract algebra in CS [closed]

31

22

What are some applications of abstract algebra in computer science an undergraduate could begin exploring after a first course?

Gallian's text goes into Hamming distance, coding theory, etc., I vaguely recall seeing discussions of abstract algebra in theory of computation / automata theory but what else? I'm not familiar with any applications past this.

Bonus: What are some textbooks / resources that one could learn about said applications?

abstract-algebra computer-science book-recommendation applications

share cite improve this question

edited Dec 31 '16 at 8:41
Martin Sleziak
43.6k ●6 ■113 ▲260

asked Dec 30 '16 at 23:30
user237975

closed as too broad by user223391, Vidyanshu Mishra, JonMark Perry, user91500, Alex Mathers Jan 6 '17 at 8:48

Please edit the question to limit it to a specific problem with enough detail to identify an adequate answer. Avoid asking multiple distinct questions at once. See the How to Ask page for help clarifying this question.

If this question can be reworded to fit the rules in the help center, please edit the question.

WHAT IS . . .

a Gröbner Basis?

Bernd Sturmfels

A *Gröbner basis* is a set of multivariate polynomials that has desirable algorithmic properties. Every set of polynomials can be transformed into a Gröbner basis. This process generalizes three familiar techniques: *Gaussian elimination* for solving linear systems of equations, the *Euclidean algorithm* for computing the greatest common divisor of two univariate polynomials, and the *Simplex Algorithm* for linear programming; see [3]. For example, the input for Gaussian elimination is a collection of linear forms such as

$$\mathcal{F} = \left\{ 2x + 3y + 4z - 5,\ 3x + 4y + 5z - 2 \right\},$$

and the algorithm transforms $\mathcal{F}$ into the Gröbner basis

$$\mathcal{G} = \left\{ \underline{x} - z + 14,\ \underline{y} + 2z - 11 \right\}.$$

Let K be any field, such as the real numbers $K = \mathbb{R}$, the complex numbers $K = \mathbb{C}$, the rational numbers $K = \mathbb{Q}$, or a finite field $K = \mathbb{F}_p$. We write $K[x_1, \dots, x_n]$ for the ring of polynomials in n variables x_i with coefficients in the field K. If $\mathcal{F}$ is any set of polynomials, then the *ideal generated* by $\mathcal{F}$ is the set $\langle \mathcal{F} \rangle$ consisting of all polynomial linear combinations:

$$\langle \mathcal{F} \rangle = \Big\{ p_1 f_1 + \cdots + p_r f_r \,:\, f_1, \dots, f_r \in \mathcal{F} \quad \text{and} \quad p_1, \dots, p_r \in K[x_1, \dots, x_n] \Big\}.$$

In our example the set $\mathcal{F}$ and its Gröbner basis $\mathcal{G}$ generate the same ideal: $\langle \mathcal{G} \rangle = \langle \mathcal{F} \rangle$. By *Hilbert's Basis Theorem*, every ideal I in $K[x_1, \dots, x_n]$ has the form $I = \langle \mathcal{F} \rangle$; i.e., it is generated by some finite set $\mathcal{F}$ of polynomials.

A *term order* on $K[x_1, \dots, x_n]$ is a total order $\prec$ on the set of all monomials $x^a = x_1^{a_1} \cdots x_n^{a_n}$ which has the following two properties:

(1) It is multiplicative; i.e., $x^a \prec x^b$ implies $x^{a+c} \prec x^{b+c}$ for all $a, b, c \in \mathbb{N}^n$.

(2) The constant monomial is the smallest; i.e., $1 \prec x^a$ for all $a \in \mathbb{N}^n \backslash \{0\}$.

Bernd Sturmfels is a professor of mathematics and computer science at the University of California at Berkeley. His email address is `bernd@math.berkeley.edu`.

An example of a term order (for $n = 2$) is the *degree lexicographic order*

$$1 \prec x_1 \prec x_2 \prec x_1^2 \prec x_1 x_2 \prec x_2^2 \prec x_1^3 \prec x_1^2 x_2 \prec \cdots.$$

If we fix a term order $\prec$, then every polynomial f has a unique *initial term* $in_\prec(f) = x^a$. This is the $\prec$-largest monomial x^a which occurs with nonzero coefficient in the expansion of f. We write the terms of f in $\prec$-decreasing order, and we often underline the initial term. For instance, a quadratic polynomial is written

$$f = \underline{3x_2^2} + 5x_1x_2 + 7x_1^2 + 11x_1 + 13x_2 + 17.$$

Suppose now that I is an ideal in $K[x_1, \dots, x_n]$. Then its *initial ideal* $in_\prec(I)$ is the ideal generated by the initial terms of all the polynomials in I:

$$in_\prec(I) = \langle\, in_\prec(f) : f \in I \,\rangle.$$

A finite subset $\mathcal{G}$ of I is a *Gröbner basis* with respect to the term order $\prec$ if the initial terms of the elements in $\mathcal{G}$ suffice to generate the initial ideal:

$$in_\prec(I) = \langle\, in_\prec(g) : g \in \mathcal{G} \,\rangle.$$

There is no minimality requirement for being a Gröbner basis. If $\mathcal{G}$ is a Gröbner basis for I, then any finite subset of I that contains $\mathcal{G}$ is also a Gröbner basis. To remedy this nonminimality, we say that $\mathcal{G}$ is a *reduced Gröbner basis* if

(1) for each $g \in \mathcal{G}$, the coefficient of $in_\prec(g)$ in g is 1,

(2) the set $\{in_\prec(g) : g \in \mathcal{G}\}$ minimally generates $in_\prec(I)$, and

(3) no trailing term of any $g \in \mathcal{G}$ lies in $in_\prec(I)$.

With this definition, we have the following theorem: If the term order $\prec$ is fixed, then every ideal I in $K[x_1, \dots, x_n]$ has a unique reduced Gröbner basis.

The reduced Gröbner basis $\mathcal{G}$ can be computed from any generating set of I by a method that was introduced in Bruno Buchberger's 1965 dissertation. Buchberger named his method after his advisor, Wolfgang Gröbner. With hindsight, the idea of Gröbner bases can be traced back to earlier sources, including a paper written in 1900 by the invariant theorist Paul Gordan. But Buchberger was the first to give an algorithm for computing Gröbner bases.

Gröbner bases are very useful for solving systems of polynomial equations. Suppose $K \subseteq \mathbb{C}$, and let $\mathcal{F}$ be a finite set of polynomials in $K[x_1, \dots, x_n]$. The *variety* of $\mathcal{F}$ is the set of all common complex zeros:

$$\mathcal{V}(\mathcal{F}) = \Big\{ (z_1, \dots, z_n) \in \mathbb{C}^n : f(z_1, \dots, z_n) = 0 \quad \text{for all} \quad f \in \mathcal{F} \Big\}.$$

The variety does not change if we replace $\mathcal{F}$ by another set of polynomials that generates the same ideal in $K[x_1, \dots, x_n]$. In particular, the reduced Gröbner basis $\mathcal{G}$ for the ideal $\langle \mathcal{F} \rangle$ specifies the same variety:

$$\mathcal{V}(\mathcal{F}) = \mathcal{V}(\langle \mathcal{F} \rangle) = \mathcal{V}(\langle \mathcal{G} \rangle) = \mathcal{V}(\mathcal{G}).$$

The advantage of $\mathcal{G}$ is that it reveals geometric properties of the variety that are not visible from $\mathcal{F}$. The first question that one might ask about a variety $\mathcal{V}(\mathcal{F})$ is whether it is empty. *Hilbert's Nullstellensatz* implies that

$$\text{the variety } \mathcal{V}(\mathcal{F}) \text{ is empty if and only if } \mathcal{G} \text{ equals } \{1\}.$$

How can one count the number of zeros of a given system of equations? To answer this, we need one more definition. Given a fixed ideal I in $K[x_1, \dots, x_n]$ and a term order $\prec$, a monomial $x^a = x_1^{a_1} \cdots x_n^{a_n}$ is called *standard* if it is not in the initial ideal $in_\prec(I)$. The number of standard monomials is finite if and only if every variable x_i appears to some power in the initial ideal. For example, if $in_\prec(I) = \langle x_1^3, x_2^4, x_3^5 \rangle$, then there are sixty standard monomials, but if $in_\prec(I) = \langle x_1^3, x_2^4, x_1 x_3^4 \rangle$, then the set of standard monomials is infinite.

The variety $\mathcal{V}(I)$ is finite if and only if the set of standard monomials is finite, and the number of standard monomials equals the cardinality of $\mathcal{V}(I)$, when zeros are counted with multiplicity. For $n = 1$ this is the *Fundamental Theorem of Algebra*, which states that the variety $\mathcal{V}(f)$ of a univariate polynomial $f \in K[x]$ of degree d consists of d complex numbers. Here the singleton $\{f\}$ is a Gröbner basis, and the standard monomials are $1, x, x^2, \dots, x^{d-1}$.

Our criterion for deciding whether a variety is finite generalizes to the following formula for the *dimension of a variety*. Consider a subset S of the variables $\{x_1, \dots, x_n\}$ such that no monomial in the variables in S appears in $in_\prec(I)$, and suppose that S has maximal cardinality among all subsets with this property. That maximal cardinality $|S|$ equals the dimension of $\mathcal{V}(I)$.

The set of standard monomials is a K-vectorspace basis for the *residue ring* $K[x_1, \dots, x_n]/I$. The image of a polynomial p modulo I can be expressed uniquely as a K-linear combination of standard monomials. This expression is the *normal form* of p. The process of computing the normal form is the *division algorithm*. In the familar case of only one variable x, where $I = \langle f \rangle$ and f has degree d, the division algorithm writes any polynomial $p \in K[x]$ as a K-linear combination of $1, x, x^2, \dots, x^{d-1}$. But the division algorithm works relative to any Gröbner basis $\mathcal{G}$ in any number of variables.

How can we test whether a given set of polynomials $\mathcal{G}$ is a Gröbner basis or not? Consider any two polynomials g and g' in $\mathcal{G}$, and form their *S-polynomial* $m'g - mg'$. Here m and m' are monomials of smallest possible degree such that $m' \cdot in_\prec(g) = m \cdot in_\prec(g')$. The S-polynomial $m'g - mg'$ lies in the ideal $\langle \mathcal{G} \rangle$. We apply the division algorithm with respect to the tentative Gröbner basis $\mathcal{G}$ to $m'g - mg'$. The resulting normal form is a K-linear combination of monomials none of which is divisible by an initial monomial from $\mathcal{G}$. A necessary condition for $\mathcal{G}$ to be a Gröbner basis is

$$\text{normalform}_{\mathcal{G}}(m'g - mg') = 0 \quad \text{for all } g, g' \in \mathcal{G}.$$

Buchberger's Criterion states that this necessary condition is sufficient: a set $\mathcal{G}$ of polynomials is a Gröbner basis if and only if all its S-polynomials have normal form zero. From this criterion, one derives *Buchberger's Algorithm* [1] for computing the reduced Gröbner basis $\mathcal{G}$ from any given input set $\mathcal{F}$.

In summary, Gröbner bases and the Buchberger Algorithm for finding them are fundamental notions in algebra. They furnish the engine for more advanced computations in algebraic geometry, such as elimination theory, computing cohomology, resolving singularities, etc. Given that polynomial models are ubiquitous across the sciences and engineering, Gröbner bases have been used by researchers in optimization, coding, robotics, control theory, statistics, molecular biology, and many other fields. We invite the reader to experiment with one of the many implementations of Buchberger's algorithm (e.g., in CoCoA, `Macaulay2`, `Magma`, `Maple`, `Mathematica`, or `Singular`).

References

[1] David Cox, John Little, and Donal O' Shea, *Ideals, Varieties and Algorithms. An Introduction to Computational Algebraic Geometry and Commutative Algebra*, second ed., Undergraduate Texts in Mathematics, Springer-Verlag, New York, 1997.

[2] Niels Lauritzen, *Concrete Abstract Algebra: From Numbers to Gröbner Bases*, Cambridge University Press, 2003.

[3] Bernd Sturmfels, *Two Lectures on Gröbner Bases*, New Horizons in Undergraduate Mathematics, VMath Lecture Series, Mathematical Sciences Research Institute, Berkeley, California, 2005, `http://www.msri.org/communications/vmath/special_productions/`.

Membership operators

- A value is in a list or not

Membership Operators	Examples	Result
`in`	`10 in (10, 20, 30)`	`True`
	`red in ('red','green','blue')`	`True`
`not in`	`10 not in (10, 20, 30)`	`False`

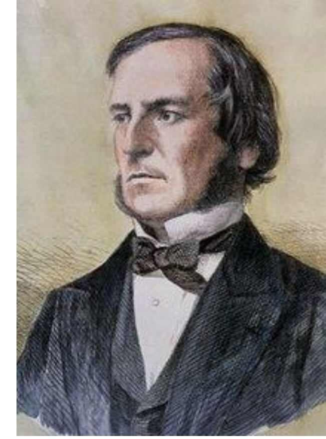
LET'S TRY IT

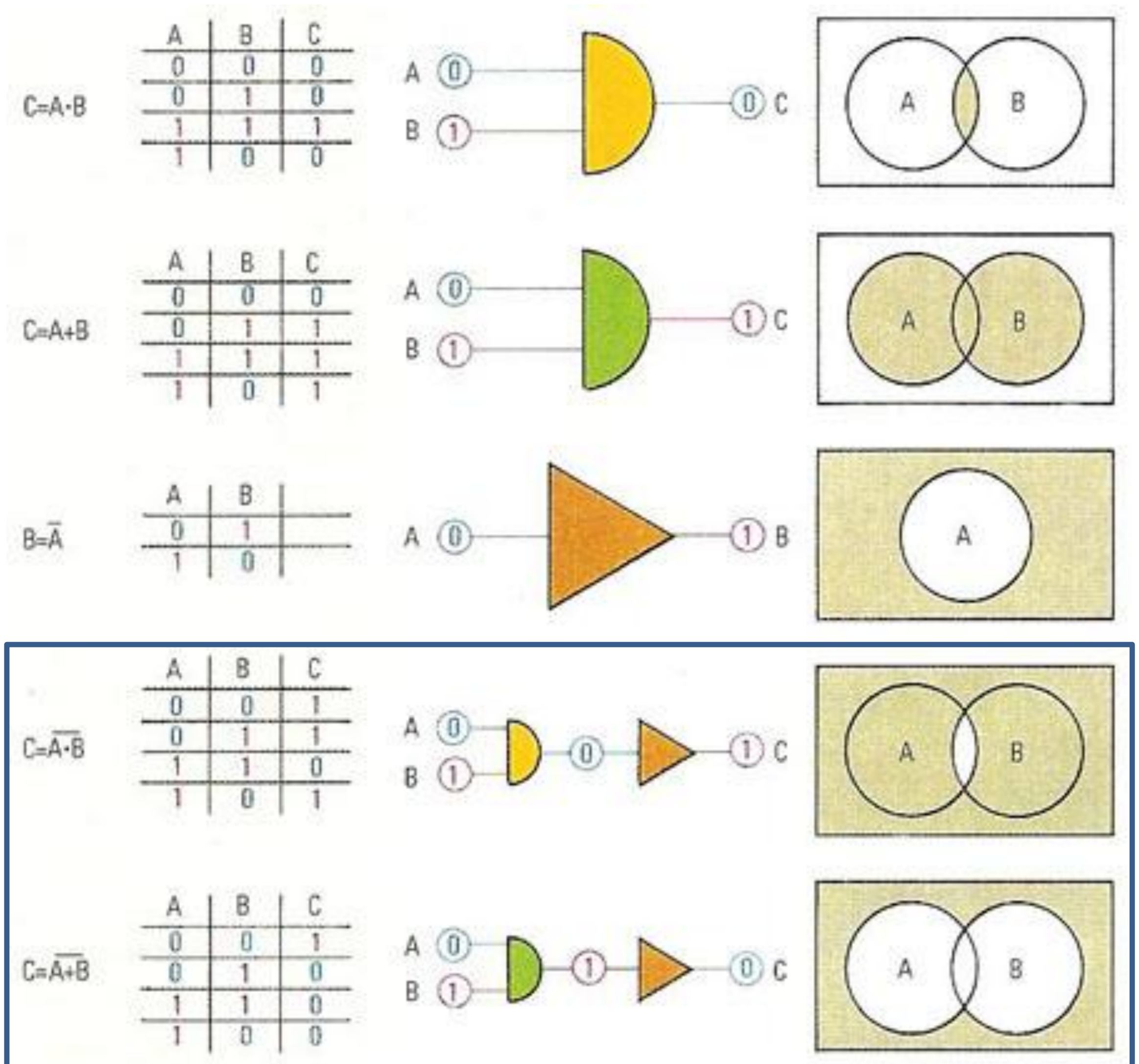
From the Python Shell, enter the following and observe the results.

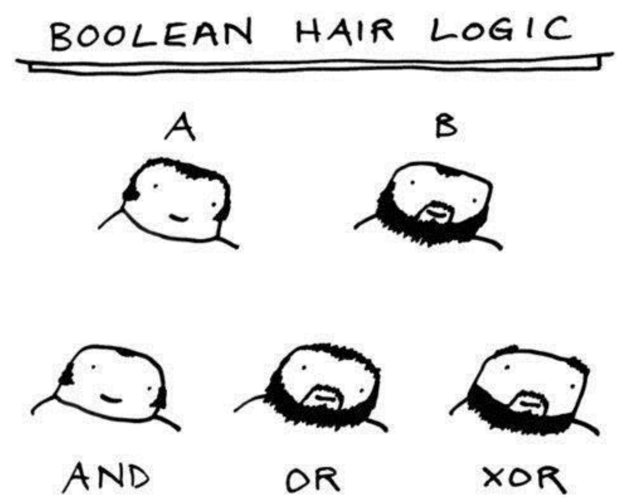
```
>>> 10 in (40, 20, 10)
???

>>> 10 not in (40, 20, 10)
???

>>> .25 in (.45, .25, .65)
???
```

```
>>> grade = 'A'
>>> grade in ('A','B','C','D','F')
???

>>> city = 'Houston'
>>> city in ('NY', 'Baltimore', 'LA')
???
```

Boolean operators







Determining intervals

1 <= num <=10

Open Interval: $(a, b) = \{x | a < x < b\}$

Close Interval: $[a,b] = \{x | a \leq x \leq b\}$

Open-close Interval: $(a,b] = \{x | a < x \leq b\}$

Close-open Interval: $[a, b) = \{x | a \leq x < b\}$

$\underbrace{\texttt{1 <= num}}\ \texttt{<= 10} \rightarrow \underbrace{\texttt{1 <= 15}}\ \texttt{<= 10} \rightarrow \texttt{True <= 10} \rightarrow \texttt{?!?}$

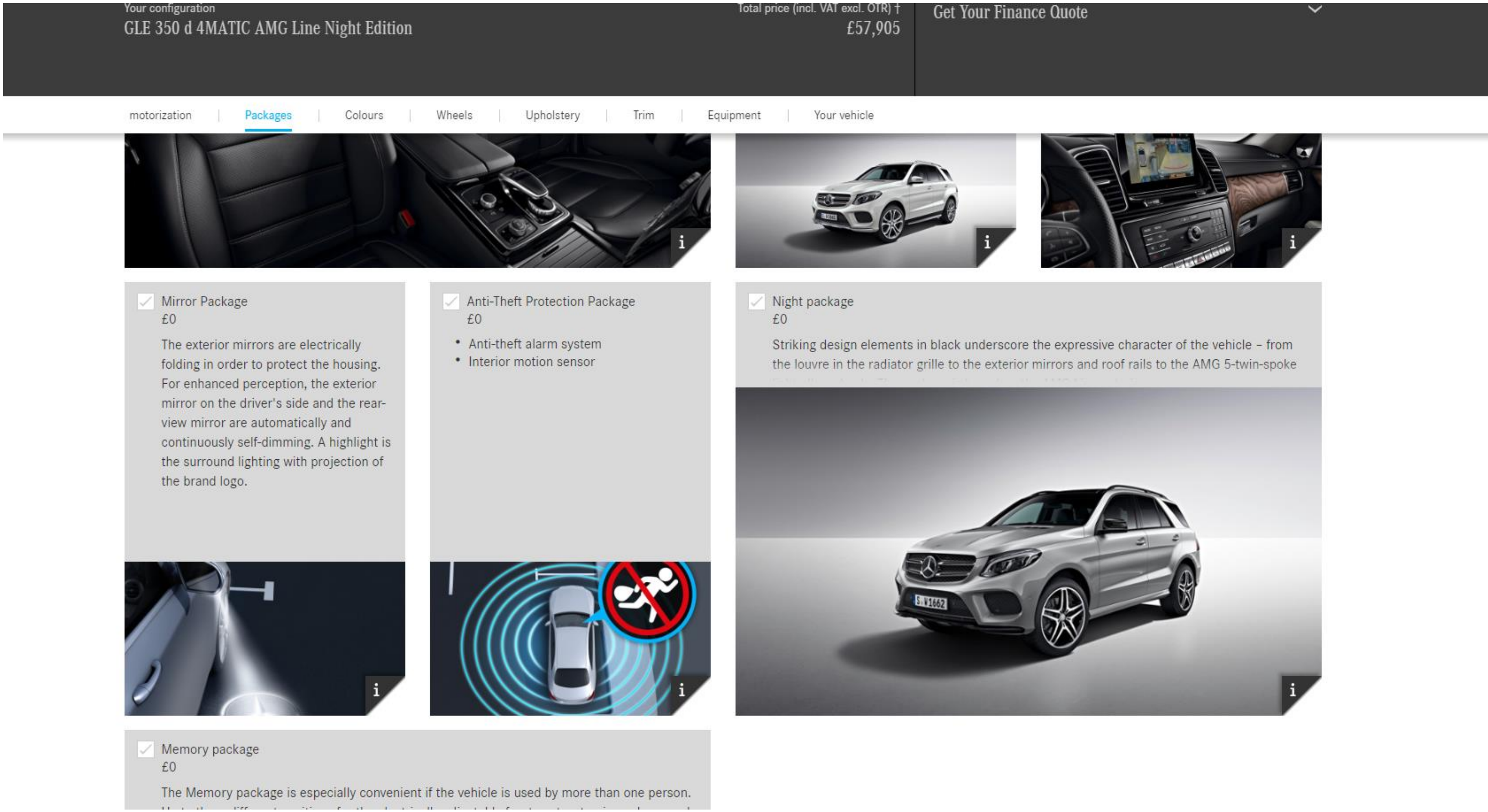
LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> True and False
???
>>> True or False
???
>>> not(True) and False
???
>>> not(True and False)
???
```

```
>>> (10 < 0) and (10 > 2)
???
>>> (10 < 0) or (10 > 2)
???
>>> not(10 < 0) or (10 > 2)
???
>>> not(10 < 0 or 10 > 2)
???
```

Product configuration



http://www.carstensinz.de/talks/RISC-2005.pdf

Product configuration

Options available for Mercedes-Benz's C class: (excerpt, total: 692)

231 garage door opener integrated into interior mirror
280 steering wheel in leather design (two-colored) with chrome clip
550 trailer appliance
581 comfort air-conditioning THERMOTRONIC
671 light metal wheels 4x, 7 spoke design
6
7
9

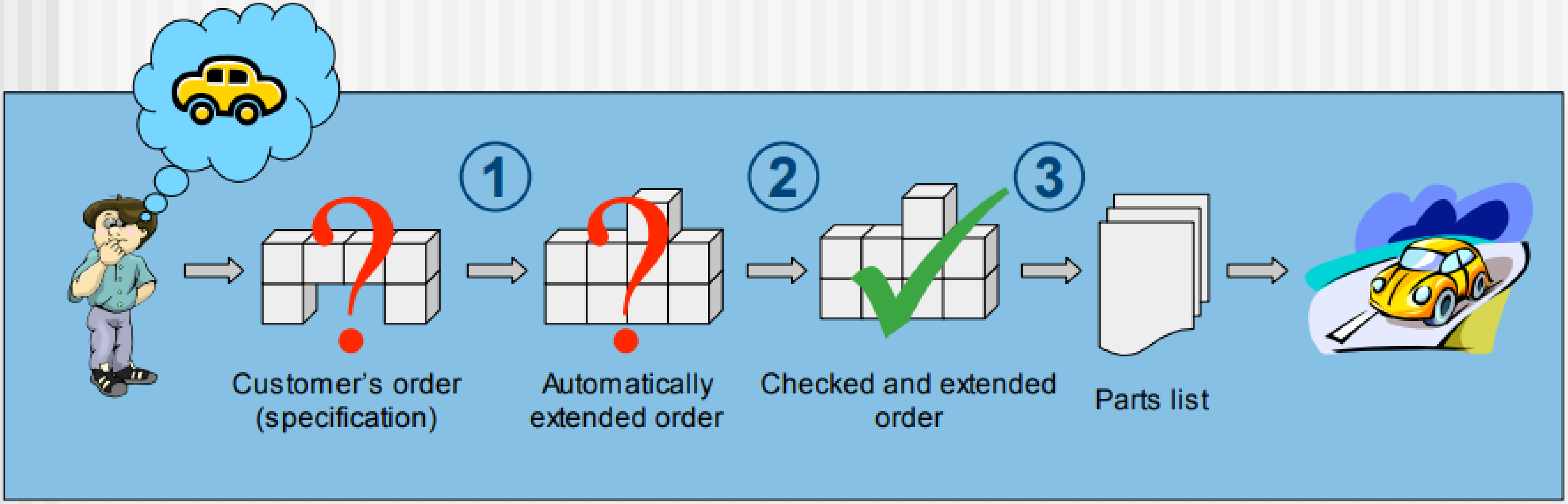
Restrictions for Mercedes-Benz's C class: (excerpt, total: 952)

AMG styling (772) cannot be combined with trailer appliance (550).

Comfort air-conditioning (581) requires high-capacity battery (673), except when combined with gasoline engines with 2.6 or 3.2 liter cylinder capacity.

http://www.carstensinz.de/talks/RISC-2005.pdf

Order processing scheme



① Order completion („supplementation“)

② Consistency check

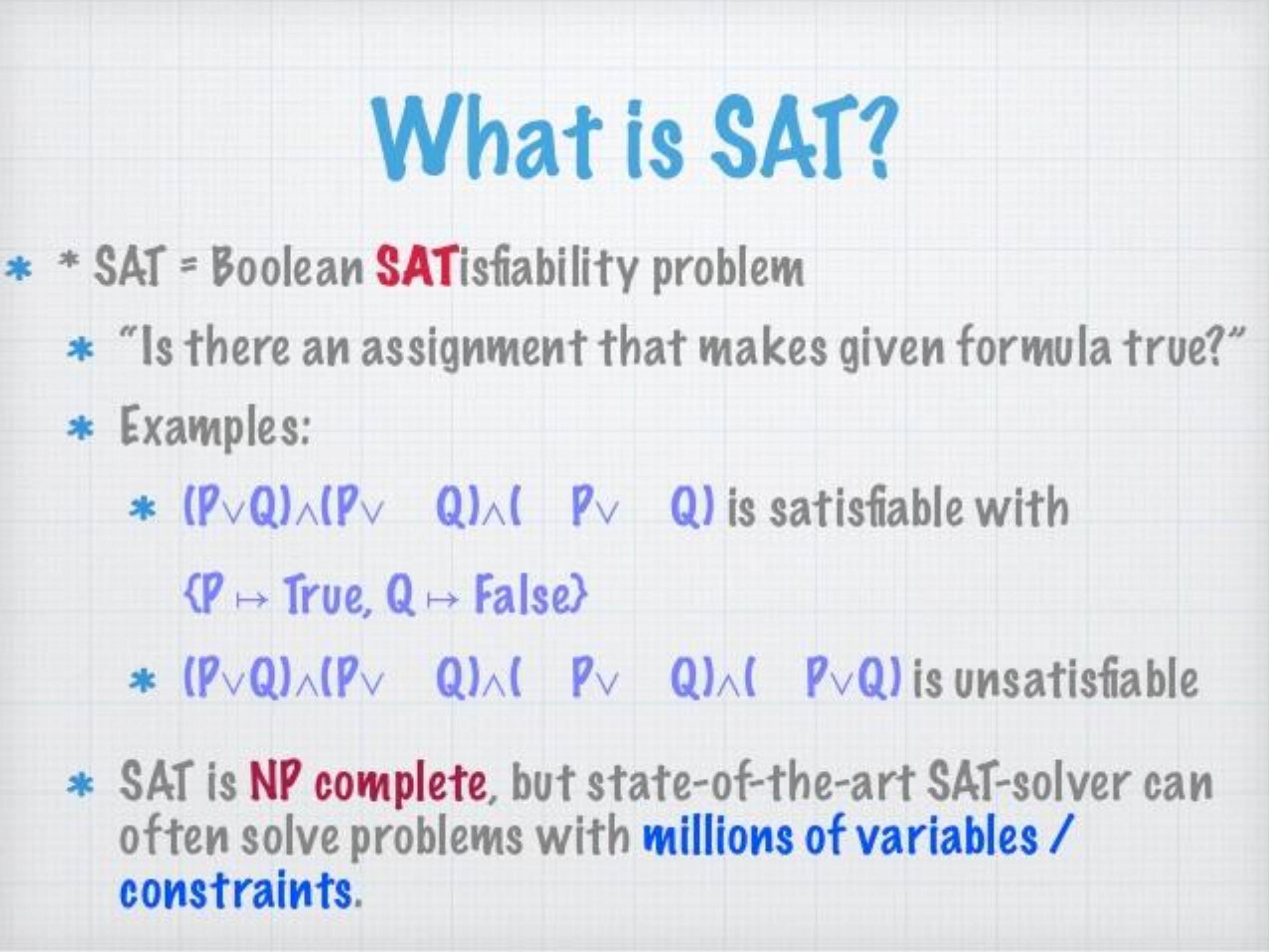
③ Generation of parts list

Product configuration

- Configurable products, model lines
  - Products assembled out of standardized components
  - E.g. computers, cars, telecommunication equipment
- Dependencies between components
  - Specified using logical formalism („*product overview*“)
- Automatic (rule-based) order processing system
  - Checks customer's order, transforms it into a parts list
- Computational problems:
  1. Determine valid (*constructible*) product instance satisfying
     - component dependencies
     - customer's restrictions
  2. Check consistency of product overview

http://www.carstensinz.de/talks/RISC-2005.pdf

SAT



http://sat.inesc-id.pt/~ines/sac10.pdf

SAT

- Boolean formula φ is defined over a set of propositional variables $x_1, \ldots, x_n$, using the standard propositional connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, and parenthesis
  - The domain of propositional variables is $\{0, 1\}$
  - Example: $\varphi(x_1, \ldots, x_3) = ((\neg x_1 \wedge x_2) \vee x_3) \wedge (\neg x_2 \vee x_3)$
- A formula φ in conjunctive normal form (CNF) is a conjunction of disjunctions (clauses) of literals, where a literal is a variable or its complement
  - Example: $\varphi(x_1, \ldots, x_3) = (\neg x_1 \vee x_3) \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee x_3)$
- Can encode any Boolean formula into CNF (more later)

http://sat.inesc-id.pt/~ines/sac10.pdf

SAT

- The Boolean satisfiability (SAT) problem:
  - Find an assignment to the variables $x_1, \ldots, x_n$ such that $\varphi(x_1, \ldots, x_n) = 1$, or prove that no such assignment exists

- SAT is an NP-complete decision problem [Cook'71]
  - SAT was the first problem to be shown NP-complete
  - There are no known polynomial time algorithms for SAT
  - 39-year old conjecture:
    Any algorithm that solves SAT is exponential in the number of variables, in the worst-case

www.satcompetition.org

- The purpose of the competition is to identify new challenging **benchmarks** and to promote new **solvers** for the propositional satisfiability problem (SAT) as well as to compare them with state-of-the-art solvers. We strongly encourage people thinking about SAT-based techniques in their area (planning, hardware or software verification, etc.) to submit benchmarks to be used for the competition. The result of the competition will be a good indicator of the current feasibility of such approach. The competition will be completely automated using the **SAT-Ex** system

The international SAT Competitions web page

Current Competition

SAT 2018 Competition	
Organizers	Marijn Heule, Matti Järvisalo, Martin Suda

Past Competitions

SAT 2017 Competition	
Organizers	Marijn Heule, Matti Järvisalo, Tomáš Balyo
Slides	Slides used at SAT 2017
Proceedings	Descriptions of the solvers and benchmarks
Benchmarks	Available here
Solvers	Available here

	Agile Track: Gold	Agile Track: Silver	Agile Track: Bronze	Main Track: Gold	Main Track: Silver	Main Track: Bronze	Random Track: Gold	Random Track: Silver	Random Track: Bronze
SAT+UNSAT	CaDiCaL Agile, CaDiCaL NoProof	Glu_VC	Glucose 4.1	Maple LCM Dist, Maple LCM, MapleLRB LCMOccRestart, MapleLRB LCM	MapleCOMSPS LRB VSIDS 2, MapleCOMSPS LRB VSIDS	COMiniSATPS Pulsar	YalSAT	tch glucose3	Score2SAT
	Parallel Track			**No-Limit Track**			**Incremental Library Track**		
SAT+UNSAT	Syrup24, Syrup48	Plingeling	Painless MapleCOMSPS	COMiniSATPS Pulsar	MapleCOMPSPS LRB VSIDS 2, MapleCOMPSPS LRB VSIDS	CaDiCaL NoProof	AbcdSAT	Glucose	Riss

SAT 2016 Competition	
Organizers	Marijn Heule, Matti Järvisalo Tomáš Balyo
Proceedings	Descriptions of the solvers and benchmarks
Benchmarks	Available here
Solvers	Available here

	Agile Track: Gold	Agile Track: Silver	Agile Track: Bronze	Main Track: Gold	Main Track: Silver	Main Track: Bronze	Random Track: Gold	Random Track: Silver	Random Track: Bronze
SAT+UNSAT	Riss	TB_Glucose	CHBR_Glucose	MapleCOMSPS	Riss	Lingeling	Dimetheus	CSCCSat	DCCAlm
	Parallel Track			**No-Limit Track**			**Incremental Library Track**		
SAT+UNSAT	Treengeling	Plingeling	CryptoMiniSat	BreakIDCOMiniSatPS	Lingeling	abcdSAT	CryptoMiniSat	Glucose	Riss
	Best Application Benchmark Solver in the Main Track			**Best Crafted Benchmark Solver in the Main Track**			**Best Glucose Hack in the Main Track**		
SAT+UNSAT	MapleCOMSPS			TC Glucose			Kiel		

SAT-Race 2015	
Organizing committee	Tomas Balyo, Carsten Sinz, Markus Iser, Armin Biere

Operator asociativity ()

Python Operators Precedence

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Ccomplement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Why is operator precedence required?

```
>>> 10 + 20 * 30 / 10
70
>>> (10 + 20) * 30 / 10
90
>>> 10 + (20 * 30) / 10
70
>>> (10 + 20 * 30) / 10
61
>>> 10 + 20 * (30 / 10)
70
```

This one Python statement could mean many things hence we need **operator precedence** to determine the **order** in which evaluation takes place

Short circuit

- In **short-circuit (lazy) evaluation**, the second operand of Boolean operators *and* and *or* is not evaluated if the value of the Boolean expression can be determined from the first operand alone.

```
if n != 0 and 1/n < tolerance:
```

Logically equivalent boolean expressions

$$(A+B)(A+C) = AA+AC+AB+BC = A(1+B+C)+BC = A+BC$$

A	B	C	A+B	A+C	(A+B)(A+C)	BC	A+BC
0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	1	1	1	1
1	0	0	1	1	1	0	1
1	0	1	1	1	1	0	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

Logically Equivalent Boolean Expressions		
`x < y`	is equivalent to	`not(x >= y)`
`x <= y`	is equivalent to	`not(x > y)`
`x == y`	is equivalent to	`not(x != y)`
`x != y`	is equivalent to	`not(x == y)`
`not(x and y)`	is equivalent to	`(not x) or (not y)`
`not(x or y)`	is equivalent to	`(not x) and (not y)`

Selection control statement

- An **if statement** is a selection control statement based on the value of a given Boolean expression.

if statement	Example use	

`if condition:`
`    statements`
`else:`
`    statements`
	`if grade >= 70:`
	`   print('passing grade')`
	`else:`
	`   print('failing grade')`
		`if grade == 100:`
		`    print('perfect score!')`

Indentation

First **clause** of if statement

```
if which == 'F':
```
header

```
    converted_temp = (temp - 32) * 5/9
    print(temp, 'degrees Fahrenheit equals',
          converted_temp, 'degrees Celsius')
```
suite

Second **clause** of if statement

```
else:
```
header

```
    converted_temp = (9/5 * temp) + 32

    print(temp, 'degrees Celsius equals',
          converted_temp, 'degrees Fahrenheit')
```
suite

Compound statements

Valid indentation		Invalid indentation	
(a)	(b)	(c)	(d)

(a)
```
 if condition:
     statement
     statement
else:
     statement
     statement
```

(b)
```
if condition:
    statement
    statement
else:
      statement
      statement
```

(c)
```
if condition:
    statement
    statement
  else:
     statement
     statement
```

(d)
```
if condition:
    statement
    statement
else:
    statement
     statement
```

LET'S TRY IT

From IDLE, create and run a Python program containing the code on the left and observe the results. Modify and run the code to match the version on the right and again observe the results. Make sure to indent the code exactly as shown.

```
grade = 90

if grade >= 70:
    print('passing grade')
else:
    print('failing grade')
```

```
grade = 90

if grade >= 70:
    print('passing grade')
  else:
    print('failing grade')
```

Nested if

Nested if statements	Example use

<pre>if *condition*:
    statements
else:
    if *condition*:
        statements
    else:
        if *condition*:
            statements
 	
            etc.</pre>
	<pre>if grade >= 90:
	    print('Grade of A')
	else:
	    if grade >= 80:
	        print('Grade of B')
	    else:
	        if grade >= 70:
	            print('Grade of C')
	        else:
	            if grade >= 60:
	                print('Grade of D')
	            else:
	                print('Grade of F')</pre>

Elif

```
if grade >= 90:
    print('Grade of A')
elif grade >= 80:
    print('Grade of B')
elif grade >= 70:
    print('Grade of C')
elif grade >= 60:
    print('Grade of D')
else:
    print('Grade of F')
```