# 1.1 Intro

**Computation** is a general term for any type of information processing. It is a process following a well-defined model that is understood and can be expressed in an algorithm, protocol, network topology, etc.

**Data** is defined as a representation of facts, concepts, or instructions in a formalized manner, suitable for communication, interpretation, or processing by a human or a computer

**Information** is the processed data that

1. is accurate and timely
2. is specific and organized for a purpose
3. is presented within a context that gives it meaning and relevance
4. can lead to an increase in understanding and decrease in uncertainty

**Information science** is a scientific field that studies information systems and information-related activities such as information: analysis, collection, classification, storage, transfer, manipulation, dissemination and protection

**Informatics** is an applied form of information science

The key characteristics of **information systems** are its abilities to acquire, transform and exchange information. They are important because they are the base requirements of any technological advancement. Information systems don’t have to be on a computer

**Computer science** is a set of skills and technologies focusing on development, programming and use of computers. Computer science deals with all practical activities related to information

A **computer** is a device that manipulates information, or data.

The fundamental operations of every computer are its ability to:

1. input information
2. store information
3. process information
4. output information

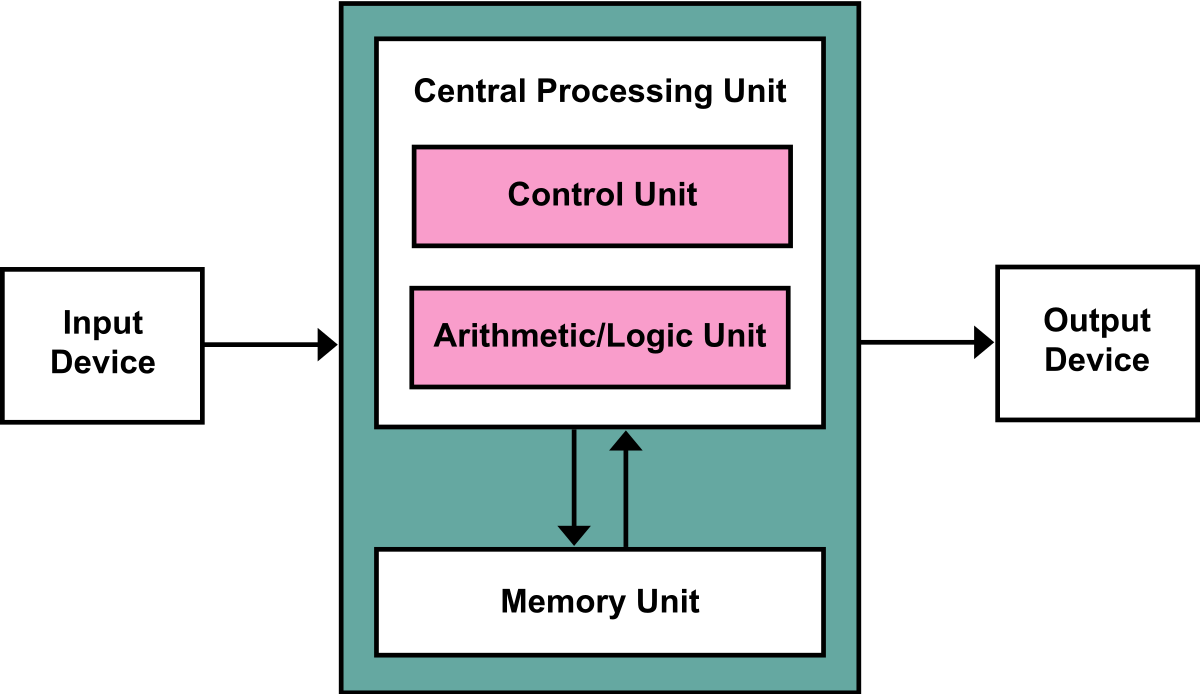
**Fixed-program computer**

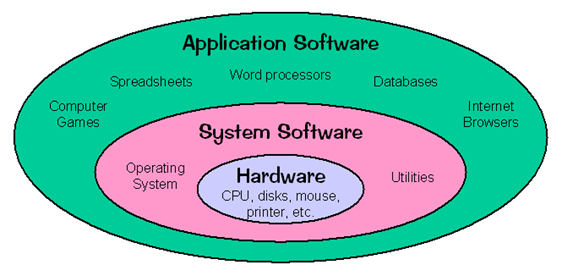
* designed to do very specific things
* changing the program requires changing the machine

**Stored-program computer**

* stores (and manipulates) a sequence of instructions, and has a set of elements that will execute any instruction in that sequence
* by treating those instructions in the same way as data, a stored-program machine can easily change the program

**von Neumann architecture (stored-program computer)**





**Analog signal** are continuous while **digital signals** are always in “steps”.

**Digitization** is the process of converting analog to digital signals (information).

Using digitization we prevent the loss of information. As a consequence, more space and time is required to store and transmit the same information.

Humans natively store and manipulate information in analog form.

Computers natively store and manipulate information in digital form.

The analog form is native for humans and as such will always be present. The digital form is likely to dominate in the future. Analog used to be more accurate, consumed less space and could be transmitted faster than the digital form, but with improvement of the technology that difference diminished. The digital information is resistant to noise and can be copied without quality degradation.

# 

# 1.2 Algorithms and Python

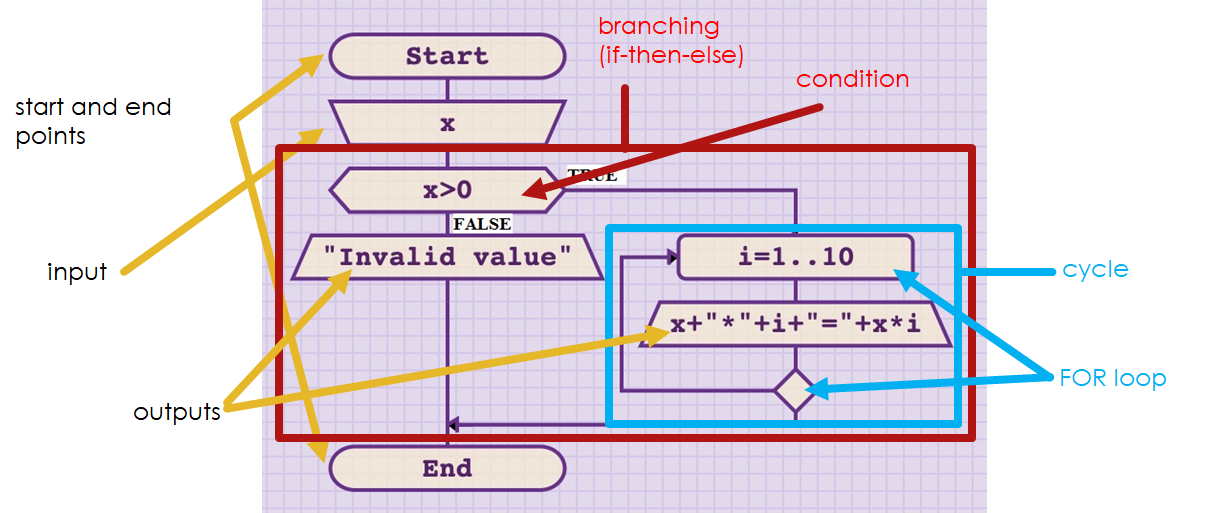
**Algorithm**

* precisely detailed step-by-step process
* used to solve problem / reach certain goal
* in a finite number of steps

Examples: cooking recipes, to-do lists

**Computer programs** are the realization of algorithms on the computer. Programmers “tell” the computer what to do and how to do it

**Flowcharts** are one way to represent an algorithm



**Algorithm categorization**

1. **linear**: every step is executed once
2. **branching**: some steps will be executed once, other steps never
3. **cyclic**: some steps can be executed multiple times2.1 Numeral systems

In our history, we have used **non-positional** and **positional numeral systems** to represent numerical values

In a **non-positional numeral system**, each symbol represents the same value regardless of its position. The numerical value is determined through the addition of values

We (humans) natively use the **decimal system**. The main reason is the number of fingers on our hands, 5 + 5 = 10. That is why early symbols were all values divisible by 5 and 10

The **decimal positional numeral system** (Hindu-Arabic, or Indo-Arabic) uses 10 symbols and the value of each symbol is determined by its face and place value. Thanks to positional numeral systems, mankind could easily express large numbers without the need for addition (and subtraction).

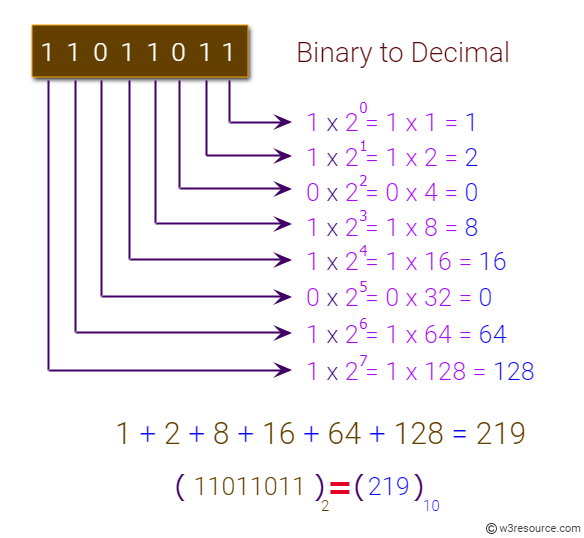
**Zero** has a twofold purpose: as placeholder and no value.

The **base** is the number of different symbols – digits (and letters) that a system of counting is using to represent numbers.

Machines cannot understand the notion of numbers. At best they can differentiate between two states: circuit open and circuit closed. That is equivalent to base 2 or the **binary** numeral system.

The **hexadecimal** and **octal** systems are used because binary numbers are not always practical (the sequence can get very long). The base for hexadecimal is 16 and for octal is 8. Each hexadecimal numeral corresponds to 4 binary bits, and each octal numeral corresponds to 3 binary bits.

Converting from **binary (base 2) to decimal**:



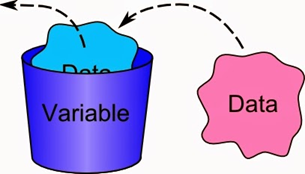
Replace 2^x with any other number to convert **from any other base**

**decimal to any other base**:

1. create string numberAsBinary
2. Use floor division to divide the number with target base
3. Store the result and the remainder
4. Add remainder to the front of numberAsBinary
5. If result is not ZERO, go back to step 2 and repeat the process with result as the new input

In Python:

|  |
| --- |
| number = 123 numberAsBinary = '' base = 2  n = number **while** n > 0:  result = n // base  remainder = n % 2  numberAsBinary = str(remainder) + numberAsBinary  n = result print(numberAsBinary) |



# 2.2 Variables and linear algorithms

Purpose variable: 

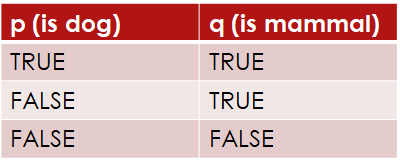
Variables can change, the stored data can’t, it’s only replaced:

# 

# 3.1 Boolean logic

|  |  |  |  |
| --- | --- | --- | --- |
| **NOT**  inverts boolean | ~ or ¬ |  |  |
| **AND**  only gives True if  both are True | ∧ |  |  |
| **OR**  gives True if one or more are True | ∨ |  |  |
| **NOR**  Only True if both are False | ↓ |  |  |
| **XOR**  only True if exactly one is True | ⊕ |  |  |
| **Implication**  *if-then* | → |  |  |
| **Equality**  *If and only if* | ↔ |  |  |
| **Therefore** | ∴ |  | - |

## Implication example



Boolean implication P implies Q ( P → Q ) means:

If P is true, then Q MUST be true.

Consequently, this implies that:

If P isn’t true, then Q can be ANYTHING.

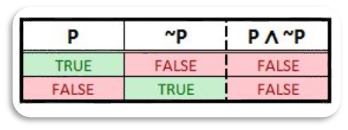
# 3.2 Branching algorithms

(many Python stuff you already know)

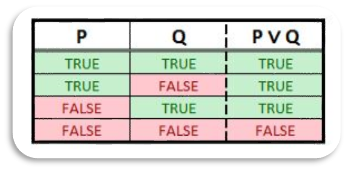
# 4.1 Logic pt2

## Tautology

A formula or assertion that is true in every possible interpretation (always TRUE)

P ∨ ~P ↔ T

## Contradiction

Is an unsatisfiable statement, both through negation or affirmation (always FALSE).

P ∧ ~P ↔ F

## Contingency

A formula that is neither Tautology nor Contradiction.

P ∨ Q

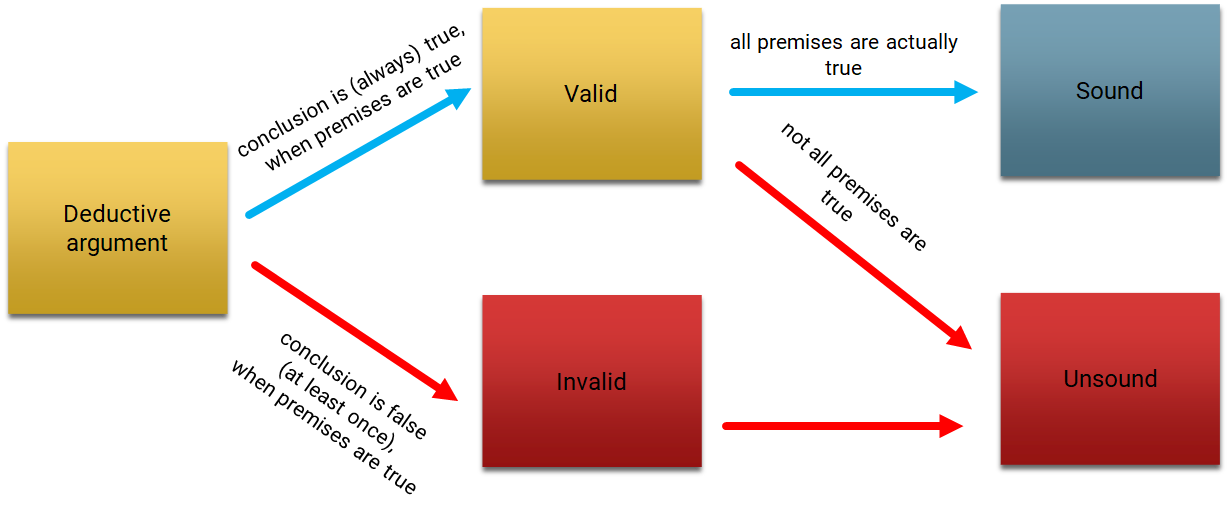
## De Morgan’s laws

Describe how mathematical statements and concepts are related through their opposites.

Use this to remove brackets

~(P ∧ Q) ↔ ~P ∨ ~Q

~(P ∨ Q) ↔ ~P ∧ ~Q



## Rules of interference

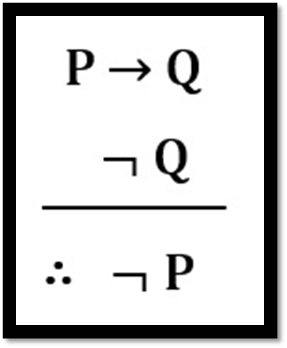
**Modus ponens** – the law of detachment

A method of affirming.

Example:

*“If it is raining, I am staying at home.”*

*“It is raining.”*

*“Therefore, I am staying at home.”*

**Modus tollens** – the law of contrapositive

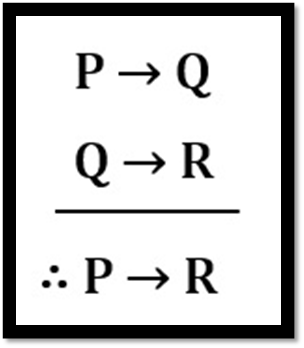
A method of denying

Example:

*‘‘If i am happy, then i am smiling.’’*

*‘’I am not smiling.’’*

*‘’Therefore, I am not happy.’’*

**Hypothetical syllogism** – chain argument

A chain argument

Example:

*‘’If I study, I will pass Analysis1.’’*

*‘’If I pass Analysis 1, I will be happy.’’*

*‘’If I study, I will be happy.’’*

# 

# 4.2 Cyclic alg

For loops & while loops, nothing you don’t know already

# 

# 5.1 Sets

**Set** = well defined collection of objects

Sets **don’t have an order**, {1, 2, 3} == {3, 1, 2} and **don’t have doubles**, {1, 2, 2, 3} == {1, 2, 3}

Use dots for an infinite list: {1, 2, 3, 4, …}

**Cardinality** = how many members the set has. Notation: n(A) or |A|

## Subset

A is **subset** of B if all elements of A are elements of B →

A is a **proper subset** if A is not equal to B

{1, 2, 3} is a subset of {1, 2, 3}, but it’s not a proper subset

## Notation

|  |  |  |
| --- | --- | --- |
| element of | ∈ | 1 ∈ {1, 2, 3} |
| not element of | ∉ | 4 ∉ {1, 2, 3} |
| contains | ∋ | {1, 2, 3} ∋ 1 |
| not contains | ∌ | {1, 2, 3} ∌ 4 |
| subset | ⊆ | {1, 2, 3} ⊆ {1, 2, 3} |
| proper subset | ⊂ or ⊊ | {1, 2} ⊂ {1, 2, 3} |
| superset | ⊇ | {1, 2, 3} ⊇ {1, 2, 3} |
| proper superset | ⊃ or ⊋ | {1, 2, 3} ⊃ {1, 2} |
| not a strict subset of | ⊄ | {1, 5} ⊄ {1, 2, 3} |
| not a strict superset of | ⊅ | {1, 2, 3} ⊄ {1, 5} |
| not a subset (nor equal to) | ⊈ | {1, 5} ⊈ {1, 2, 3} |
| not a superset (nor equal to) | ⊉ | {1, 2, 3} ⊉ {1, 5} |
| Empty set | {} or ∅ | |∅| == 0 |

Empty set is a subset of every set

**Universal set** is the set that contains everything

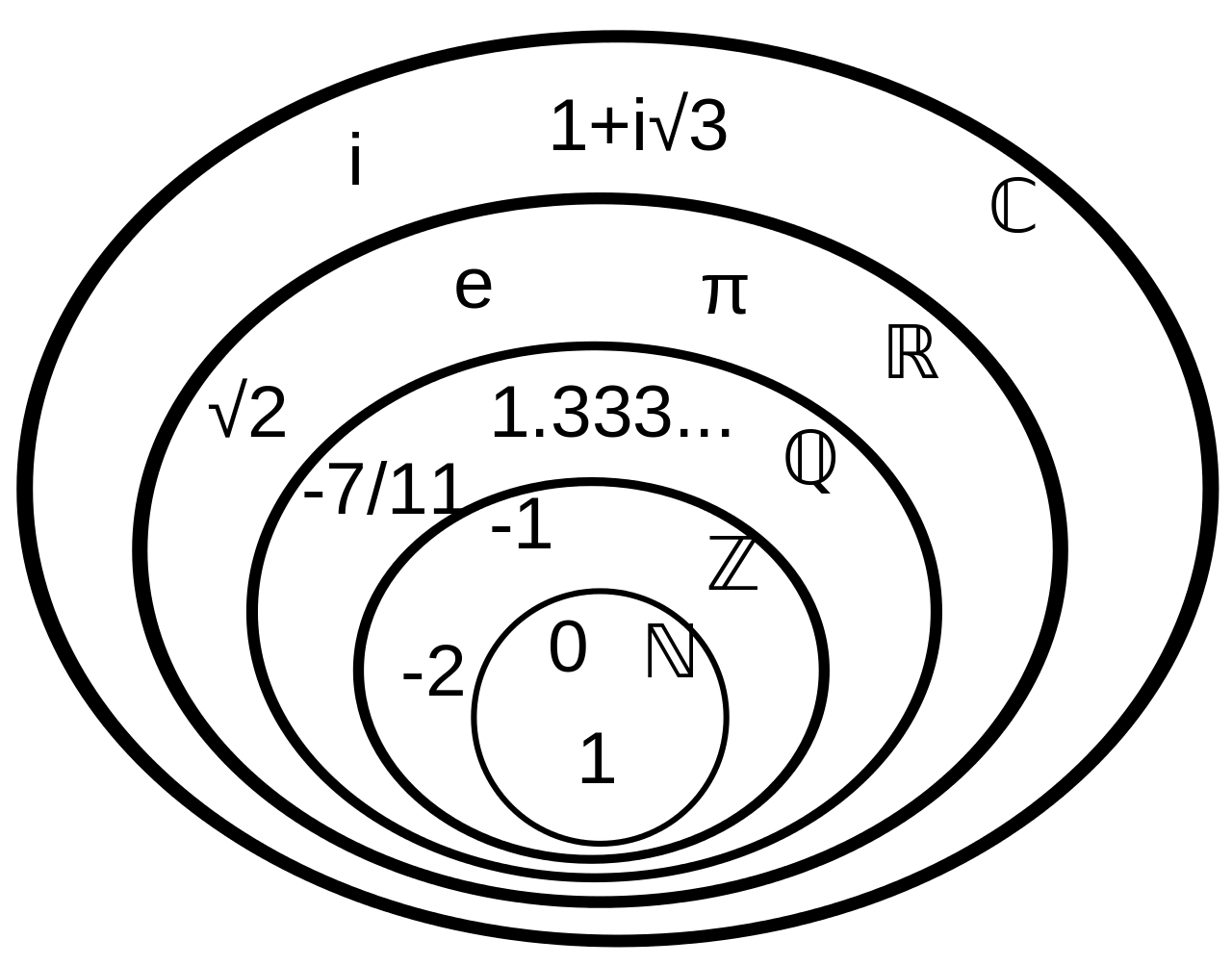
**Disjoint sets** don’t share any members

## Set operations

|  |  |
| --- | --- |
| Union A ∪ B  Combination of both sets, doubles removed  {1,2,3} ∪ {2, 4, 5, 1} == {1, 2, 3, 4, 5} |  |
| Intersection A ∩ B  Only the elements that are in both sets  {1, 2, 3} ∩ {2, 4, 5, 1} == {1, 2} |  |
| Difference A - B  All elements of A that are not in B  {1, 2, 3} - {2, 4, 5, 1} == {3} |  |
| Universal set ξ  All elements |  |
| Complement Ac or A'  All elements of the universal set that are not in A  ξ = {1, 2, 3, 4, 5, 6, 7, 8, 9}  A = {1, 2, 3}  Ac = {4, 5, 6, 7, 8, 9} |  |

## Common number sets

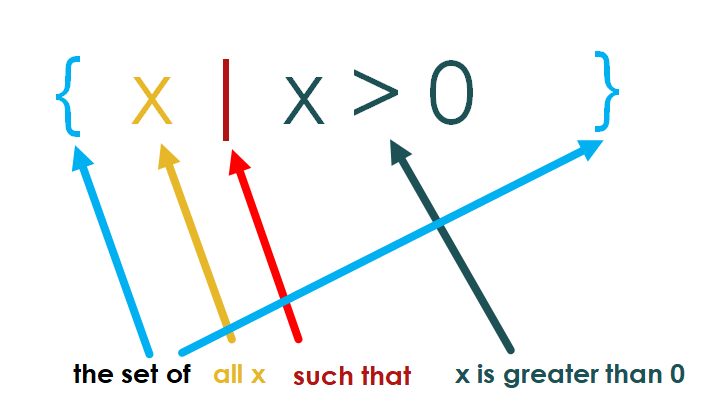
|  |  |  |  |
| --- | --- | --- | --- |
|  | **name** | **definition** | **examples** |
| ℕ | Natural numbers | All whole numbers higher than 0 | 1, 7, 666 |
| ℤ | Integers | All whole numbers including 0 | -420, 0, 69 |
| ℚ | Rational numbers | All fractions | 10/4, -1/5, 3/1, 0 |
| ℝ | Real numbers | Any number | √2, 𝜋, 0 |
| 𝕀 | Imaginary numbers | [The numbers that give a negative result when squared](https://en.wikipedia.org/wiki/Imaginary_number) | i, 2i |
| ℂ | Complex numbers | All [complex numbers](https://en.wikipedia.org/wiki/Complex_number) | 1 + 2i, 7 + 3i |



## Set-builder notation

**Set builder notation** is a way to define sets without having to write out each member

the set {1, 2, 3, …} could be written as:



All integers that are less than or equal to 10: A = {x ∈ Z | x ≤ 10 } == {..., 9, 10}

All real numbers between 1 and 8 (both including): A = {x ∈ R | x ≥ 1 and x ≤ 8 } == {1, …, 8}

## Interval

Interval is all numbers between two given numbers

( ) = **open interval** = not including the number (a,b) == a < x < b

[ ] = **closed interval** = including the number [a, b] == a <= x <= b

can mix, including a but not b: [a, b) == a <= x < b

Infinity is not a number, so always open interval: [a, +∞**)**

Use Union (U) and intersection (∩) to join multiple intervals

(-∞, -5] U (10, +∞) x ≤ -5 or x > 10

(-∞, 5] ∩ (1, ∞) x > 1 and x ≤ 5 (1, 5]

## Power set

**Power set** is a set of all subsets of a set.

Notation: P(A)

P({1}) = {{1}, ∅}

P({1, 2, 3}) = {{1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3}, ∅}

Cardinality: n(P(A)) = 2^n(A)

## Cartesian product

Also known as **cross product**

Notation: A × B

Set-builder notation: {(a, b) : a ∈ A and b ∈ B}

Match every element from B once with every element from A, the element from A always first

A = {∝, β}

B = {1, 2, 3}

A × B = {(∝, 1), (∝, 2), (∝, 3), (β, 1) , (β, 2) , (β, 3)}

Cardinality: n(A × B) = n(A) \* n(B) = 2 \* 3 = 6

C = {2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A}

D = {♠, ♥, ♦, ♣}

C × D = {2♠, 2♥, 2♦, 2♣, 3♠, 3♥, 3♦, 3♣, … , A♠, A♥, A♦, A♣}

## Cartesian coordinates

x and y coordinates

Notation: (x, y)

# 5.2 Sets and cyclic alg 2

## Sets in Python

Notation:

A = set() for an empty set

Warning: don’t use {} to create an empty set in Python, it will create a dictionary instead

A = {1, 2, 3} for a filled set

|  |  |
| --- | --- |
| A.add(4) | Adds an element to the set |
| A.clear() | Removes all elements from the set |
| B = A.copy() | Returns a copy of the set |
| A.discard(4) | Removes an element from the set (Nothing if not exists) |
| A.remove(4) | Removes an element from the set (Error if not exists) |
| A.update({4, 5, 6}) | Updates the set with the union of itself and others |
| C = A.intersection(B) | Returns the intersection of two sets as a new set |
| C = A.isdisjoint(B) | Returns True if both sets have no common elements |
| C = A.issubset(B) | Returns True if another set contains this set |
| C = A.issuperset(B) | Returns True if this set is contained in another set |
| C = A.union(B) | Returns the union of sets in a new set |
| E = max(A) | Returns the largest item in the set |
| E = min(A) | Returns the smallest item in the set |
| C = sorted(A) | Returns a new sorted list from elements in the set |
| E = sum(A) | Returns the sum of all elements in the set |

(Hier mist nog heel wat, kijk zelf even in presentatie 5.2)

# 6.1 Problem solving

## Categorical proposition

A Categorical proposition (= categorical statement) is a statement about the relationship between categories.

A categorical proposition states: whether one category is:

* fully contained in another category
* partially contained within another category
* completely separate

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name** | | **Sentence (example)** | **Standard form** | **Quantity** | **Quality** |
| **A** | **A**ffirmo | “All cats have tail.” | All S is P | universal | affirmative |
| **E** | n**E**go | “No human has wings.” | No S is P | universal | negative |
| **I** | aff**I**rmo | “Some dogs are friendly.” | Some S is P | particular | affirmative |
| **O** | neg**O** | “Some teachers are not gamers.” | Some S is not P | particular | negative |

## Representing categorical proposition with Venn diagram

|  |  |
| --- | --- |
| Shading = empty | X = some (at least 1) |
| All S are P | No S is P |
| Some S is P | Some S is not P |

(uitleggen X op een randje)

If the conclusion is contained within the appropriate area and doesn’t require adding additional information ➜ conclusion is **valid**

else ➜ conclusion is **invalid**

# 6.2 Lists

List is a mutable, ordered sequence of items

Notation:

A = [] for an empty list

A = [1, 2, 3]

To access an item, use A[index], index starts with 0:

A[0] gives 1

A[2] gives 3

Negative index counts from the end:

A[-1] gives 3

Use [start:end:step] for getting a part of the list

Start is included, end is not

You can omit any one of the three and it will use default values

[1, 2, 3, 4, 5][1:3] gives [2, 3]

[1, 2, 3, 4, 5][::2] gives

**Tuple** is same as list, but you can’t change it after you created it, it will give an error if you try

You write a tuple with normal brackets:

(1, 2, 3)

(hier mist nog heel wat, kijk zelf naar presentatie 6.2)