Introduction to Computer Programming
-
Introduction

Summer 2022

Computers

Uses of computers

- send emails
- write reports
- browse the web
- play games
- ...

Design of computers

Computers are designed to run **programs**.
Programs are sets of instructions that a computer can perform.

Computers

Hardware and Software

Physical devices that a computer is made of are referred to **hardware**
Programs are referred to **software**

A computer by itself does not perform any task, but it can run programs

Typical hardware system

- Central Processing Unit (CPU)
- Random-Access Memory (RAM)
- Storage devices (Hard drives, SSD, . . .)
- Input devices
- Output devices

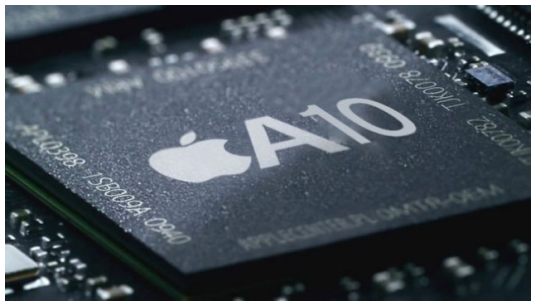
Computers

Smartphone example

A smartphone **is** a computer system

- CPU: Apple A10, Snapdragon 820, ...
- RAM: 2-4 GB RAM
- Storage: 32-128 GB (internal) + SD cards
- Input devices: Buttons, touchscreen, microphone ...
- Output devices: Screen, vibrations, speakers, ...

CPU

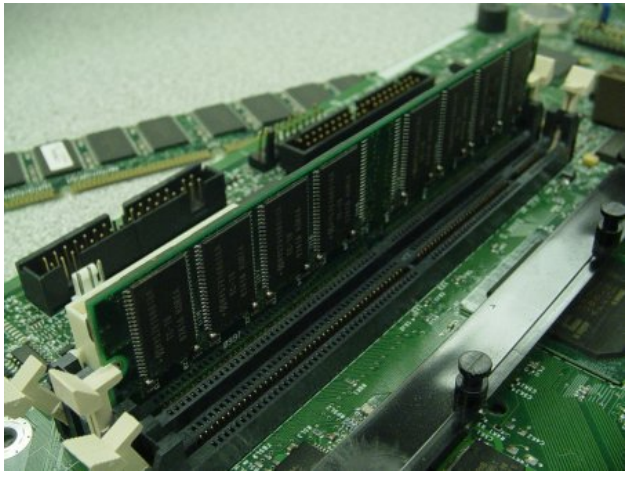


CPU

Most important part in a computer

- *Actually* runs programs
- also known as **microprocessors**
- performs operations such as addition, substraction, multiplication, . . . and more complex instructions

Random Acces Memory (RAM)



RAM

Computer's work area

- where the computer stores a running program
- . . . and the data associated to this program

- **Volatile** memory → needs power to store data
- when power is off, the contents of RAM are lost

Storage

Storage

Can hold data for long periods of time, even without power

Input devices

Input devices

Collects data from people and from other devices
Data can be:

- text
- numerical values
- sound
- video
- ...

Output devices

Output devices

Presents\produces data to people

Software

Software (program)

- Computer hardware is useless without software
- Software is the set of instructions and associated data that direct the computer to do a task

2 categories of software

- System software
  - Operating systems (Windows, Linux, MacOS, Android, iOS, ...)
  - Utility programs (antivirus, ...)
  - Software development tools (IDLE, ...)
- Application software
  - programs you usually launch on your computer/smartphone
  - example: Chrome, Powerpoint, Skype, ...

Computer data

Binary information

Computers can only handle **binary** information.
The tiniest information is a **bit** (*binary digit*, symbol **b**).
2 states for a bit: **0** (off) and **1** (on)

Reason: Electronic hardware can easily store/handle binary information

Groups of bits

A single bit can only take 2 different values: 0 or 1
A group of 2 bits can take 4 different values: 00, 01, 10 or 11
A group of 3 bits can take 8 different values: 000, 001, ...
...
A group of **n bits** can take $\mathbf{2^n}$ **different values**

Computer data

Byte

Groups of **8 bits** are called **bytes** (symbol **B**)
A byte can take **256** different values

Questions

How many different values can be represented by:

- 1 bit (1 b)?
- 2 bits (2 b)?
- 8 bits (8 b)?
- 1 byte (1 B)?
- 2 bytes (2 B)?
- 4 bytes (4 B)?

Storing integers

Integer on a single byte

Using a single byte, 256 different values can be obtained
For **positive** integers, one byte can represent integers between **0** and **255**

Integer on n bits

2^n different values can be obtained
It will represent integers between **0** and **2^n-1**

Logic

- Number 0 is obtained when every bit is 0
- The maximum integer (2^n-1) is obtained when every bit is 1

Storing integers

Binary Numbering System

Each bit represents a **power of 2**: 2^0, 2^1, 2^2, ...
Binary Numbering System is very similar to Decimal System we use

Example on one byte

1	0	1	0	1	1	0	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	**32**	16	**8**	**4**	2	**1**

Result: $128+32+8+4+1=\mathbf{173}$

Decimal

1	0	6
10^2	10^1	10^0
100	10	1

Result:
$1 \cdot 100 + 0 \cdot 10 + 6 \cdot 1 = \mathbf{106}$

Storing integers

Conversion from decimal to binary

Find the biggest powers of 2 in the number

Example: $89 = \mathbf{64} + \ldots$
$= \mathbf{64} + 25$
$= \mathbf{64} + \mathbf{16} + \ldots$
$= \mathbf{64} + \mathbf{16} + 9$
$= \mathbf{64} + \mathbf{16} + \mathbf{8} + \ldots$
$= \mathbf{64} + \mathbf{16} + \mathbf{8} + \mathbf{1}$
$= 2^6 + 2^4 + 2^3 + 2^0$

0	1	0	1	1	0	0	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Storing integers

Notation

In order to avoid confusion between numbers written in binary or decimal systems (for example 101), binary representation can be preceded by **0b**

$$0b101 = 5$$
$$101 = 0b1100100$$

Questions

Convert to decimal

- 0b00000010
- 0b11000000
- 0b01011111
- 0b11111111
- 0b10101010

Convert to binary (on one byte)

- 12
- 100
- 164
- 247
- 300

Storing numbers

Negative numbers

A slightly different strategy is usually used to store negative integers
It is called **2's complement**

not covered in this course

Decimal point

2 strategies exist to encode decimal point numbers:

- fixed point (using negative powers of 2)
- floating point

not covered in this course

Storing text

Text

Encoding numbers with bits (or bytes) is quite logical (a mathematical relationship exists)
But **how to encode text** (letters, punctuation, ...) ?

American Standard Code for Information Interchange (ASCII)

Table of correspondences between numbers (on 7 bits) and characters
$\rightarrow$ 128 characters

Strong limitations:

- Chinese characters?
- Accented letters?
- Mathematical symbols?
- Currency symbols?
- ...

Storing text

ASCII table

000	(nul)	016 ►	(dle)	032	sp	048	0	064	@	080	P	096	`	112	p
001 ☺	(soh)	017 ◄	(dc1)	033	!	049	1	065	A	081	Q	097	a	113	q
002 ☻	(stx)	018 ↕	(dc2)	034	"	050	2	066	B	082	R	098	b	114	r
003 ♥	(etx)	019 ‼	(dc3)	035	#	051	3	067	C	083	S	099	c	115	s
004 ♦	(eot)	020 ¶	(dc4)	036	$	052	4	068	D	084	T	100	d	116	t
005 ♣	(enq)	021 §	(nak)	037	%	053	5	069	E	085	U	101	e	117	u
006 ♠	(ack)	022 ▬	(syn)	038	&	054	6	070	F	086	V	102	f	118	v
007 •	(bel)	023 ↨	(etb)	039	'	055	7	071	G	087	W	103	g	119	w
008 ◘	(bs)	024 ↑	(can)	040	(	056	8	072	H	088	X	104	h	120	x
009	(tab)	025 ↓	(em)	041	)	057	9	073	I	089	Y	105	i	121	y
010	(lf)	026	(eof)	042	*	058	:	074	J	090	Z	106	j	122	z
011 ♂	(vt)	027 ←	(esc)	043	+	059	;	075	K	091	[	107	k	123	{
012 ♀	(np)	028 ∟	(fs)	044	,	060	<	076	L	092	\	108	l	124	|
013	(cr)	029 ↔	(gs)	045	-	061	=	077	M	093	]	109	m	125	}
014 ♫	(so)	030 ▲	(rs)	046	.	062	>	078	N	094	^	110	n	126	~
015 ☼	(si)	031 ▼	(us)	047	/	063	?	079	O	095	_	111	o	127	⌂

Storing text

Unicode

The current standard for consistently representing text through different character sets and encodings is called **Unicode**

- latest version covers more than 128,000 characters
- matches ASCII
- integers encode the character

https://unicode-table.com/en/

Programming

How does a program run?

The program is executed by the CPU!!!

CPU performs basic operations such as:

- add, substract, multiply, divide, ...
- read and write piece of data in memory (RAM for example)
- ...

The CPU executes instructions provided by a program

CPU Instructions

As a digital component, the CPU only understands binary. Instructions are encoded in binary (it is called instruction set).

For example, if the CPU encounters the instruction **0b01010011**, it may **add 3 to a value**

Programming

DO NOT WORRY!!!!!

We won't be programming in machine language

Several programming languages exist, it enables a programmer to write in a convenient language that will be turned *later* into machine language

Languages

Natural languages

- meant for communications between people
- it has evolved naturally (not designed)
- usually verbose
- sometimes brings ambiguity

Programming languages

- artificial
- created specifically to provide **instructions** to CPU
- very rigid structure
- **very strict on syntax**
- some words have a particular meaning/role
- is very explicit (no room for interpretation)

It provides a **layer of abstraction** between programmer and machine

Programming languages

So many programming languages

Popularity (based on TIOBE index on May 2017):

- 1. Java
- 2. C
- 3. C++
- 4. **Python**
- 5. C#

http://www.tiobe.com/tiobe-index/

Which one to use?

It depends...

Some languages are specific for some platforms (mobile for example).
Personal preference is often considered if possible.

Programming languages

Characteristics of programming languages

- High-level and low-level
- Interpreted and compiled
- *Programming paradigms*...

Low-level languages

Purpose

Designed for direct execution by the machine (only possible with low-level languages)

Examples: **Machine code** and **assembly**

Example: Assembly

```
move.b    #4,d0    ; load register d0 with value 4
add.b     #3,d0    ; add 3 to register d0
```

Example: Machine code

```
0b11100100
0b10100011
```

High-level languages

Purpose

Designed to be easy for humans to read and write

- must be **compiled** or **interpreted** into a low-level language in order to be executed
- provides abstraction that free the programmer from dealing with the underlying machine
- usually prioritizes usability over efficiency
- *portable* across different Operating Systems/Architectures

Example: Python

```
1 for i in range(0,100):
2     print(i)
```

Compiled and interpreted languages

2 approaches

- **Compiled**
  - compiles the **source code** into a **low-level** language using a **compiler**
- **Interpreted**
  - executes the **source code** immediately using an **interpreter**

Different characteristics

- Compiled languages are usually considered faster and more efficient at execution
- Interpreted languages are generally thought as easier to use (immediate feedback, . . .)

Compilation

Compiling a program

- **source code** is the program before compilation
- a **compiler** turns it into **object code** (low-level)

After compilation, the **object code** can be executed by the machine

Usual workflow

1. Write code
2. Compile
3. Execute program
4. Go back to step 1

Interpretation

Interpreting a program

- the program is executed through an **interpreter**
- the **interpreter** reads **one statement** of the source code at a time
  - it turns it into **machine language**
  - and **runs** it
- it then reads **next statement** of the source code and so on...

Usual workflow

1. Write code
2. Execute program
3. Go back to step 1

Programming workflow

Main steps

Programming can usually be divided into the following major steps:

1. Requirements gathering
2. Implementation
3. Run the program
4. Check output
5. Go back to step 2
   - or possibly even step 1!
   - do this until. . . ?

Requirements gathering

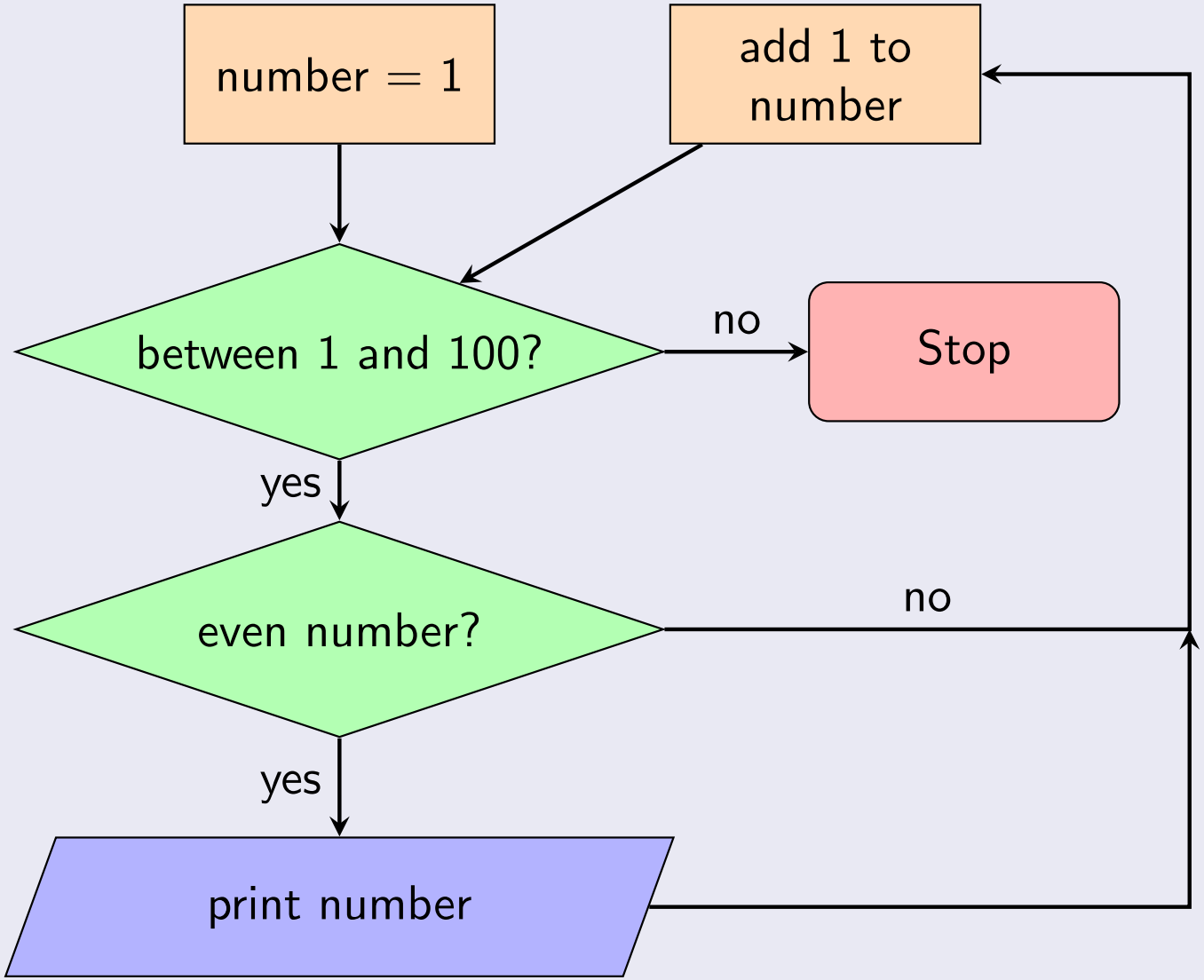
Problem definition

What your program should do?

- Problem solving
- Design techniques and documentation
  - Flowcharts, sequence diagrams, ...
  - Pseudocode

Requirements gathering

Flowchart for printing even numbers between 1 and 100



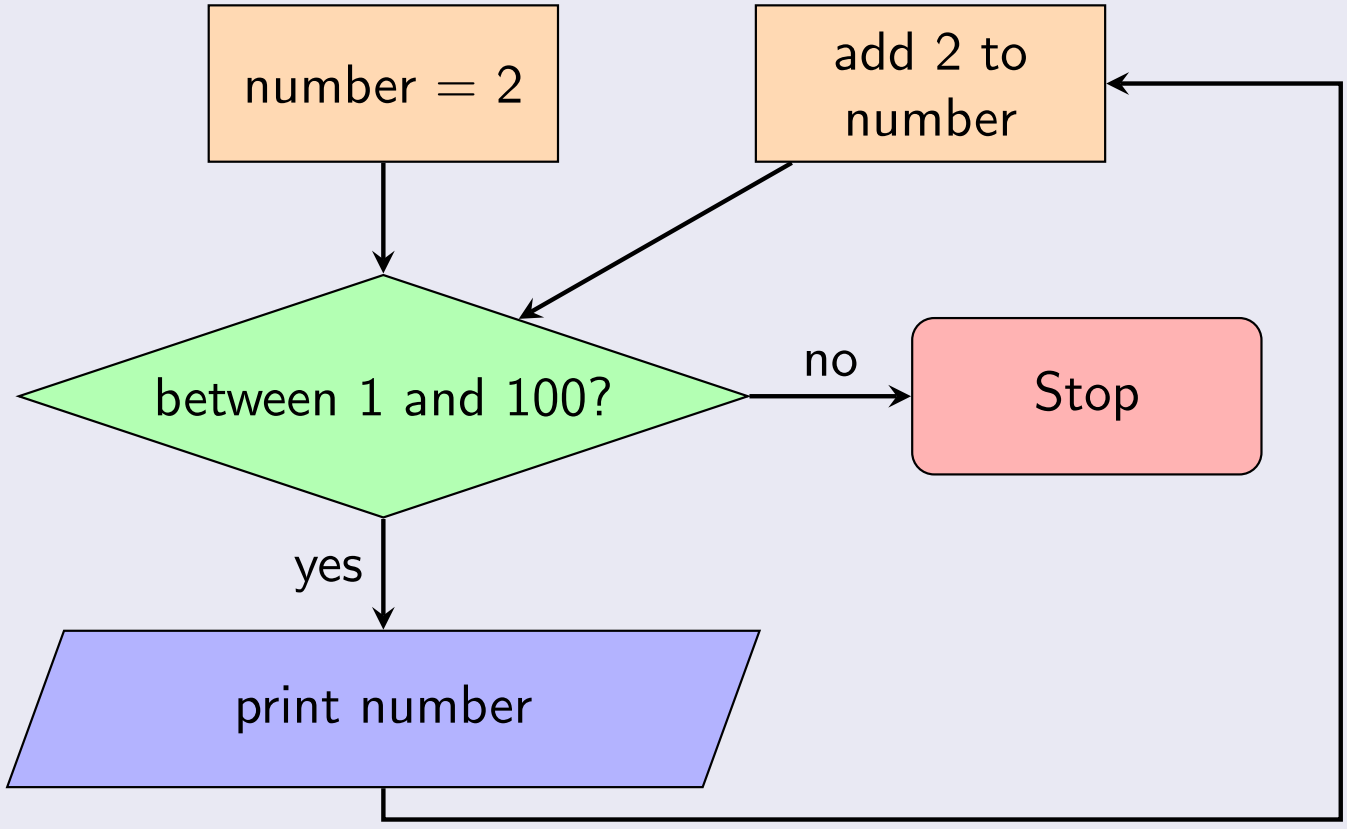
Requirements gathering

Pseudocode for printing even numbers between 1 and 100

```
Begin
        Set number to 1
        While 1≤number≤100
                If number is even
                        Print number
                Add 1 to number
End
```

Requirements gathering

Other solution: Flowchart for printing even numbers between 1 and 100



Requirements gathering

Other solution: Pseudocode for printing even numbers between 1 and 100

```
Begin
        Set number to 2
        While 1≤number≤100
                Print number
                Add 2 to number
End
```

Coding/Implementation

Implementation

- Actually writing the code
- It is not just writing new code, it can be:
  - bug-fixing (finding and removing errors)
  - refactoring (improving the code for efficiency or maintainability, . . .)

Python code for printing even numbers between 1 and 100

```
1 for number in range(2,101,2):
2     print(number)
```

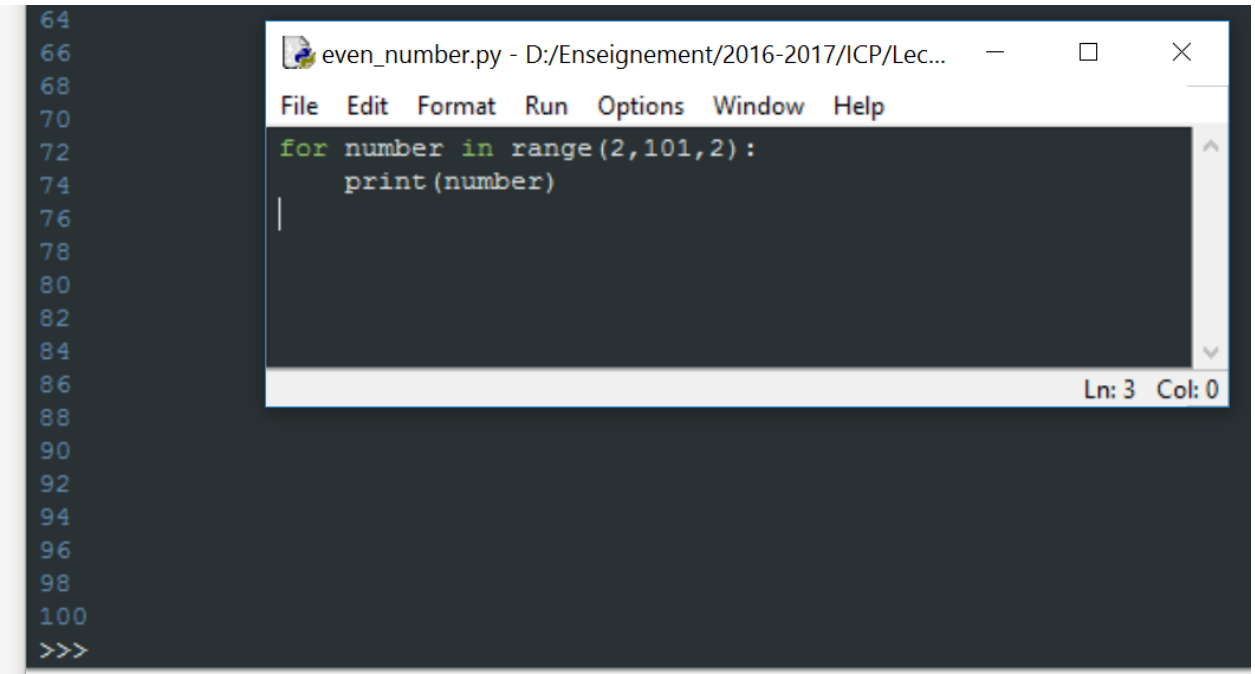
Program execution

Executing the program

Since we are using Python, an interpreted language, we do not need to compile before running the program

We can execute our program:

- by typing it line by line in the **interpreter**
- or by writing every line in a **source file** and running the code



Checking output

Check if everything was fine

- Any error message during execution?
  - **Syntax error**: problem in the syntax of the program
  - **Runtime error**: a generic error type meaning there is a problem with the program
  - Other errors: ...
- Check output result (can be printed values or a generated textfile, ...)

If the output result is incorrect, you have a **logical error**.

Debug

Debugging a program is **tracking down** the root cause of any **error** and **fixing** it
Some errors are sometimes easy to find and fix and some errors can be quite difficult to locate and/or fix

Iteration

Main steps

Programming can usually be divided into the following major steps:

1. Requirements gathering
2. Implementation
3. Run the program
4. Check output
5. Go back to step 2
   - or possibly even step 1!
   - do this until. . . ?

Iteration

Programming workflow

Programming workflow is a **loop repeated several times** until the program finally **meet the requirements**

. . . but sometimes a programmer still wants to improve his program (efficiency, better interface, . . .). Then it can become an endless workflow until the programmer is finally satisfied. . .