

COMP110: Principles of Computing

# 3: Flowcharts and pseudocode

# Learning outcomes

- ▶ Explain and use basic data types
- ▶ Recall some basic algorithms
- ▶ Produce and explain basic flowcharts
- ▶ Produce and explain basic pseudocode

# Data types



# What is a type?

# What is a type?

- ▶ A **variable** in Python holds a **value**

# What is a type?

- ▶ A **variable** in Python holds a **value**
- ▶ Every value has a **type**

# What is a type?

- ▶ A **variable** in Python holds a **value**
- ▶ Every value has a **type**
- ▶ The type of a value dictates:

# What is a type?

- ▶ A **variable** in Python holds a **value**
- ▶ Every value has a **type**
- ▶ The type of a value dictates:
  - ▶ What sort of data it can hold



# What is a type?

- ▶ A **variable** in Python holds a **value**
- ▶ Every value has a **type**
- ▶ The type of a value dictates:
  - ▶ What sort of data it can hold
  - ▶ How the data is stored in memory

# What is a type?

- ▶ A **variable** in Python holds a **value**
- ▶ Every value has a **type**
- ▶ The type of a value dictates:
  - ▶ What sort of data it can hold
  - ▶ How the data is stored in memory
  - ▶ What operations can be done on it

# What is a type?

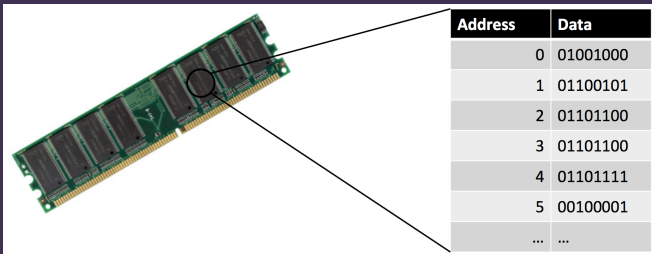
- ▶ A **variable** in Python holds a **value**
- ▶ Every value has a **type**
- ▶ The type of a value dictates:
  - ▶ What sort of data it can hold
  - ▶ How the data is stored in memory
  - ▶ What operations can be done on it
- ▶ Python is **weakly typed** — a variable can hold a value of any type

# What is a type?

- ▶ A **variable** in Python holds a **value**
- ▶ Every value has a **type**
- ▶ The type of a value dictates:
  - ▶ What sort of data it can hold
  - ▶ How the data is stored in memory
  - ▶ What operations can be done on it
- ▶ Python is **weakly typed** — a variable can hold a value of any type
- ▶ Many languages are **strongly typed** — a variable has a defined type and can only hold values of that type

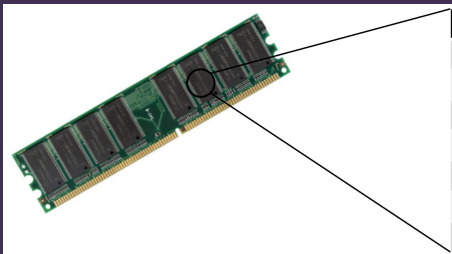
# Memory

# Memory



- ▶ Memory works like a set of **boxes**

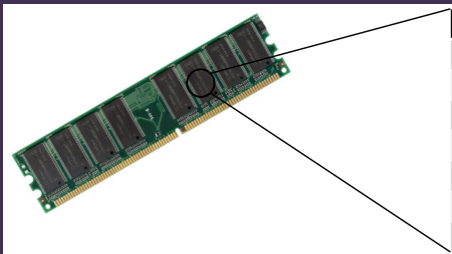
# Memory



Address	Data
0	01001000
1	01100101
2	01101100
3	01101100
4	01101111
5	00100001
...	...

- ▶ Memory works like a set of **boxes**
- ▶ Each box has a number, its **address**

# Memory



Address	Data
0	01001000
1	01100101
2	01101100
3	01101100
4	01101111
5	00100001
...	...

- ▶ Memory works like a set of **boxes**
- ▶ Each box has a number, its **address**
- ▶ Each box contains a **byte** (8 bits)



# Data representation

# Data representation

- ▶ All data is stored as **sequences of bytes**

# Data representation

- ▶ All data is stored as **sequences of bytes**
  - ▶ Sequence of bits, in multiples of 8

# Data representation

- ▶ All data is stored as **sequences of bytes**
  - ▶ Sequence of bits, in multiples of 8
  - ▶ Sequence of numbers between 0–255

# Integers

# Integers

- ▶ An **integer** is a whole number — positive, negative or zero

# Integers

- ▶ An **integer** is a whole number — positive, negative or zero
- ▶ Python type: `int`

# Integers

- ▶ An **integer** is a whole number — positive, negative or zero
- ▶ Python type: `int`
- ▶ In most languages, `int` is limited to 32 or 64 bits



# Integers

- ▶ An **integer** is a whole number — positive, negative or zero
- ▶ Python type: `int`
- ▶ In most languages, `int` is limited to 32 or 64 bits
- ▶ Python uses **big integers** — number of bits expands automatically to fit the value to be stored

# Integers

- ▶ An **integer** is a whole number — positive, negative or zero
- ▶ Python type: `int`
- ▶ In most languages, `int` is limited to 32 or 64 bits
- ▶ Python uses **big integers** — number of bits expands automatically to fit the value to be stored
- ▶ Stored in memory using binary notation, with 2's complement for negative values

# Floating point numbers

# Floating point numbers

- ▶ What about storing non-integer numbers?

# Floating point numbers

- ▶ What about storing non-integer numbers?
- ▶ Usually we use **floating point** numbers

# Floating point numbers

- ▶ What about storing non-integer numbers?
- ▶ Usually we use **floating point** numbers
- ▶ Python type: `float`

# Floating point numbers

- ▶ What about storing non-integer numbers?
- ▶ Usually we use **floating point** numbers
- ▶ Python type: `float`
- ▶ Details on in-memory representation later in the module

# Floating point numbers

- ▶ What about storing non-integer numbers?
- ▶ Usually we use **floating point** numbers
- ▶ Python type: `float`
- ▶ Details on in-memory representation later in the module
- ▶ (Note: `float` in Python 3 has the same precision as `double` in C++/C#/etc)



# Integers vs floating point numbers

# Integers vs floating point numbers

- ▶ `int` and `float` are different types!

# Integers vs floating point numbers

- ▶ `int` and `float` are different types!
- ▶ 42 and 42.0 are technically different values

# Integers vs floating point numbers

- ▶ `int` and `float` are different types!
- ▶ 42 and 42.0 are technically different values
  - ▶ One is an `int`, the other is a `float`

# Integers vs floating point numbers

- ▶ `int` and `float` are different types!
- ▶ 42 and 42.0 are technically different values
  - ▶ One is an `int`, the other is a `float`
  - ▶ They are stored differently in memory

# Integers vs floating point numbers

- ▶ `int` and `float` are different types!
- ▶ `42` and `42.0` are technically different values
  - ▶ One is an `int`, the other is a `float`
  - ▶ They are stored differently in memory
  - ▶ However `==` etc still know how to compare them sensibly

# Booleans

# Booleans

- ▶ A **boolean** can have one of two values: **true** or **false**



# Booleans

- ▶ A **boolean** can have one of two values: **true** or **false**
- ▶ Python type: `bool`

# Booleans

- ▶ A **boolean** can have one of two values: **true** or **false**
- ▶ Python type: `bool`
- ▶ In Python, we have the keywords `True` and `False`

# Booleans

- ▶ A **boolean** can have one of two values: **true** or **false**
- ▶ Python type: `bool`
- ▶ In Python, we have the keywords `True` and `False`
- ▶ Could be represented by a single bit in memory...

# Booleans

- ▶ A **boolean** can have one of two values: **true** or **false**
- ▶ Python type: `bool`
- ▶ In Python, we have the keywords `True` and `False`
- ▶ Could be represented by a single bit in memory...
- ▶ ... but since memory is addressed in bytes (or words of multiple bytes), usually represented as an `int` with 0 meaning `False` and any non-zero (e.g. 1) meaning `True`

# Boolean values

# Boolean values

- ▶ The `if` statement takes a boolean value as its condition:

```
if x > 10:  
    print(x)
```

# Boolean values

- ▶ The `if` statement takes a boolean value as its condition:

```
if x > 10:  
    print(x)
```

- ▶ Variables can also store boolean values:

```
result = (x > 10)    # result now stores True or False  
if result:  
    print(x)
```

# The “None” value



# The “None” value

- ▶ Python has a special value `None` which can be used to denote the “absence” of any other value

# The “None” value

- ▶ Python has a special value `None` which can be used to denote the “absence” of any other value
- ▶ Python type: `NoneType`

# Strings

# Strings

- ▶ A **string** represents a sequence of textual characters

# Strings

- ▶ A **string** represents a sequence of textual characters
- ▶ E.g. `"Hello world!"`

# Strings

- ▶ A **string** represents a sequence of textual characters
- ▶ E.g. "Hello world!"
- ▶ Python type: `str`

# Strings

# Strings

- ▶ Stored as sequences of **characters** encoded as **integers**



# Strings

- ▶ Stored as sequences of **characters** encoded as **integers**
- ▶ Often **null-terminated**

# Strings

- ▶ Stored as sequences of **characters** encoded as **integers**
- ▶ Often **null-terminated**
  - ▶ Character number 0 signifies the end of the string

# Strings

- ▶ Stored as sequences of **characters** encoded as **integers**
- ▶ Often **null-terminated**
  - ▶ Character number 0 signifies the end of the string
- ▶ **ASCII** encodes characters as 8-bit integers

# Strings

- ▶ Stored as sequences of **characters** encoded as **integers**
- ▶ Often **null-terminated**
  - ▶ Character number 0 signifies the end of the string
- ▶ **ASCII** encodes characters as 8-bit integers
  - ▶ 255 characters: Latin alphabet, numerals, punctuation

# Strings

- ▶ Stored as sequences of **characters** encoded as **integers**
- ▶ Often **null-terminated**
  - ▶ Character number 0 signifies the end of the string
- ▶ **ASCII** encodes characters as 8-bit integers
  - ▶ 255 characters: Latin alphabet, numerals, punctuation
- ▶ **UTF-32** encodes characters as 32-bit integers

# Strings

- ▶ Stored as sequences of **characters** encoded as **integers**
- ▶ Often **null-terminated**
  - ▶ Character number 0 signifies the end of the string
- ▶ **ASCII** encodes characters as 8-bit integers
  - ▶ 255 characters: Latin alphabet, numerals, punctuation
- ▶ **UTF-32** encodes characters as 32-bit integers
  - ▶ **Unicode** characters: ASCII + several other alphabets, Asian languages, symbols, emoji, ...

# Strings

- ▶ Stored as sequences of **characters** encoded as **integers**
- ▶ Often **null-terminated**
  - ▶ Character number 0 signifies the end of the string
- ▶ **ASCII** encodes characters as 8-bit integers
  - ▶ 255 characters: Latin alphabet, numerals, punctuation
- ▶ **UTF-32** encodes characters as 32-bit integers
  - ▶ **Unicode** characters: ASCII + several other alphabets, Asian languages, symbols, emoji, ...
- ▶ **UTF-8** encodes characters as 8, 16, 24 or 32-bit integers

# Strings

- ▶ Stored as sequences of **characters** encoded as **integers**
- ▶ Often **null-terminated**
  - ▶ Character number 0 signifies the end of the string
- ▶ **ASCII** encodes characters as 8-bit integers
  - ▶ 255 characters: Latin alphabet, numerals, punctuation
- ▶ **UTF-32** encodes characters as 32-bit integers
  - ▶ **Unicode** characters: ASCII + several other alphabets, Asian languages, symbols, emoji, ...
- ▶ **UTF-8** encodes characters as 8, 16, 24 or 32-bit integers
  - ▶ More common Unicode characters are smaller  $\implies$  more efficient than UTF-32



# Decimal - Binary - Octal - Hex – ASCII Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	\$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

# String representation

# String representation

- ▶ `"Hello world!"` in ASCII encoding:

# String representation

► "Hello world!" in ASCII encoding:

72	101	108	108	111	32	119	111	114	108	100	33	0
----	-----	-----	-----	-----	----	-----	-----	-----	-----	-----	----	---

# UTF-8 representation

# UTF-8 representation

- ▶ For characters in ASCII, UTF-8 is the same:

# UTF-8 representation

- ▶ For characters in ASCII, UTF-8 is the same:
  - ▶  $a \rightarrow [97]$

# UTF-8 representation

- ▶ For characters in ASCII, UTF-8 is the same:
  - ▶  $a \rightarrow [97]$
- ▶ Other characters are encoded as multi-byte sequences:



# UTF-8 representation

- ▶ For characters in ASCII, UTF-8 is the same:
  - ▶  $a \rightarrow [97]$
- ▶ Other characters are encoded as multi-byte sequences:
  - ▶  $\ddot{u} \rightarrow [195, 188]$

# UTF-8 representation

- ▶ For characters in ASCII, UTF-8 is the same:
  - ▶  $a \rightarrow [97]$
- ▶ Other characters are encoded as multi-byte sequences:
  - ▶  $\ddot{u} \rightarrow [195, 188]$
  - ▶ 串  $\rightarrow [228, 184, 178]$

# UTF-8 representation

- ▶ For characters in ASCII, UTF-8 is the same:
  - ▶ a → [97]
- ▶ Other characters are encoded as multi-byte sequences:
  - ▶ ü → [195, 188]
  - ▶ 串 → [228, 184, 178]
  - ▶ 🤔 → [240, 159, 152, 130]

# Type casting

# Type casting

- It is often useful to **cast**, or **convert**, a value from one type to another

# Type casting

- ▶ It is often useful to **cast**, or **convert**, a value from one type to another
- ▶ In Python, this is done by calling the type as if it were a function

# Type casting

- ▶ It is often useful to **cast**, or **convert**, a value from one type to another
- ▶ In Python, this is done by calling the type as if it were a function
  - ▶ `float(17)` → 17.0

# Type casting

- ▶ It is often useful to **cast**, or **convert**, a value from one type to another
- ▶ In Python, this is done by calling the type as if it were a function
  - ▶ `float(17)` → `17.0`
  - ▶ `int(3.14)` → `3`



# Type casting

- ▶ It is often useful to **cast**, or **convert**, a value from one type to another
- ▶ In Python, this is done by calling the type as if it were a function
  - ▶ `float(17)` → `17.0`
  - ▶ `int(3.14)` → `3`
  - ▶ `str(3.14)` → `"3.14"`

# Type casting

- ▶ It is often useful to **cast**, or **convert**, a value from one type to another
- ▶ In Python, this is done by calling the type as if it were a function
  - ▶ `float(17)` → `17.0`
  - ▶ `int(3.14)` → `3`
  - ▶ `str(3.14)` → `"3.14"`
  - ▶ `str(1 + 1 == 2)` → `"True"`

# Type casting

- ▶ It is often useful to **cast**, or **convert**, a value from one type to another
- ▶ In Python, this is done by calling the type as if it were a function
  - ▶ `float(17)` → `17.0`
  - ▶ `int(3.14)` → `3`
  - ▶ `str(3.14)` → `"3.14"`
  - ▶ `str(1 + 1 == 2)` → `"True"`
  - ▶ `int("123")` → `123`

# Type casting

- ▶ It is often useful to **cast**, or **convert**, a value from one type to another
- ▶ In Python, this is done by calling the type as if it were a function
  - ▶ `float(17)` → `17.0`
  - ▶ `int(3.14)` → `3`
  - ▶ `str(3.14)` → `"3.14"`
  - ▶ `str(1 + 1 == 2)` → `"True"`
  - ▶ `int("123")` → `123`
  - ▶ `int("five")` gives an error

# Other types

# Other types

- ▶ **Container** types for collecting several values

# Other types

- ▶ **Container** types for collecting several values
  - ▶ `list`, `tuple`, `dict`, `set`, ...

# Other types

- ▶ **Container** types for collecting several values
  - ▶ `list, tuple, dict, set, ...`
- ▶ **Objects**



# Other types

- ▶ **Container** types for collecting several values
  - ▶ `list, tuple, dict, set, ...`
- ▶ **Objects**
- ▶ Almost everything in Python is a value with a type

# Other types

- ▶ **Container** types for collecting several values
  - ▶ `list`, `tuple`, `dict`, `set`, ...
- ▶ **Objects**
- ▶ Almost everything in Python is a value with a type
  - ▶ Functions, modules, classes, exceptions, ...

# Other types

- ▶ **Container** types for collecting several values
  - ▶ `list`, `tuple`, `dict`, `set`, ...
- ▶ **Objects**
- ▶ Almost everything in Python is a value with a type
  - ▶ Functions, modules, classes, exceptions, ...
  - ▶ Even types themselves are values of type `type`!

# Algorithms



# What is an algorithm?

# What is an algorithm?

A **sequence of instructions** which can be followed **step by step** to perform a **computational task**.

# Programs vs algorithms

# Programs vs algorithms

- ▶ A program is **specific** to a particular programming language and/or machine



# Programs vs algorithms

- ▶ A program is **specific** to a particular programming language and/or machine
- ▶ An algorithm is **general**

# Programs vs algorithms

- ▶ A program is **specific** to a particular programming language and/or machine
- ▶ An algorithm is **general**
- ▶ An algorithm must be **implemented** as a program before a computer can run it

# Programs vs algorithms

- ▶ A program is **specific** to a particular programming language and/or machine
- ▶ An algorithm is **general**
- ▶ An algorithm must be **implemented** as a program before a computer can run it
- ▶ An algorithm generally performs **one task**, whereas a program may perform **many**

# Programs vs algorithms

- ▶ A program is **specific** to a particular programming language and/or machine
- ▶ An algorithm is **general**
- ▶ An algorithm must be **implemented** as a program before a computer can run it
- ▶ An algorithm generally performs **one task**, whereas a program may perform **many**
  - ▶ E.g. Microsoft Word is not an algorithm, but it implements many algorithms

# Programs vs algorithms

- ▶ A program is **specific** to a particular programming language and/or machine
- ▶ An algorithm is **general**
- ▶ An algorithm must be **implemented** as a program before a computer can run it
- ▶ An algorithm generally performs **one task**, whereas a program may perform **many**
  - ▶ E.g. Microsoft Word is not an algorithm, but it implements many algorithms
  - ▶ E.g. it implements an algorithm for determining where to break a line of text, how much space to add to centre a line, etc.

# Algorithms outside computing

- 1 Preheat the oven to 180C, gas 4.
- 2 Beat together the eggs, flour, caster sugar, butter and baking powder until smooth in a large mixing bowl.
- 3 Put the cocoa in separate mixing bowl, and add the water a little at a time to make a stiff paste. Add to the cake mixture.
- 4 Turn into the prepared tins, level the top and bake in the preheated oven for about 20-25 mins, or until shrinking away from the sides of the tin and springy to the touch.
- 5 Leave to cool in the tin, then turn on to a wire rack to become completely cold before icing.
- 6 To make the icing: measure the cream and chocolate into a bowl and carefully melt over a pan of hot water over a low heat, or gently in the microwave for 1 min (600w microwave). Stir until melted, then set aside to cool a little and to thicken up.
- 7 To ice the cake: spread the apricot jam on the top of each cake. Spread half of the ganache icing on the top of the jam on one of the cakes, then lay the other cake on top, sandwiching them together.
- 8 Use the remaining ganache icing to ice the top of the cake in a swirl pattern. Dust with icing sugar to serve.

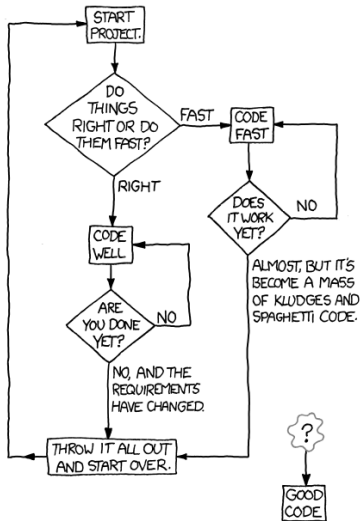
[illegible]

# Flowcharts

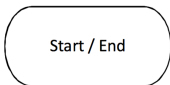




## HOW TO WRITE GOOD CODE:



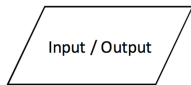
# Flowchart symbols



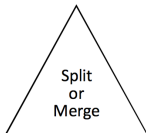
The start or end of a workflow.



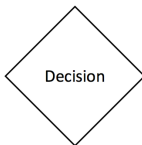
Process or action.



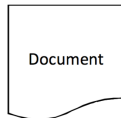
Data: Inputs to, and outputs from, a process.



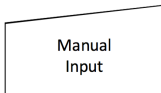
Upright indicates a process split,  
inverted indicates a merge of processes.



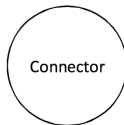
Decision point in a  
process or workflow.



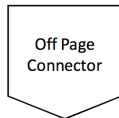
Document or report.



Prompt for information, manually  
entered into a system.

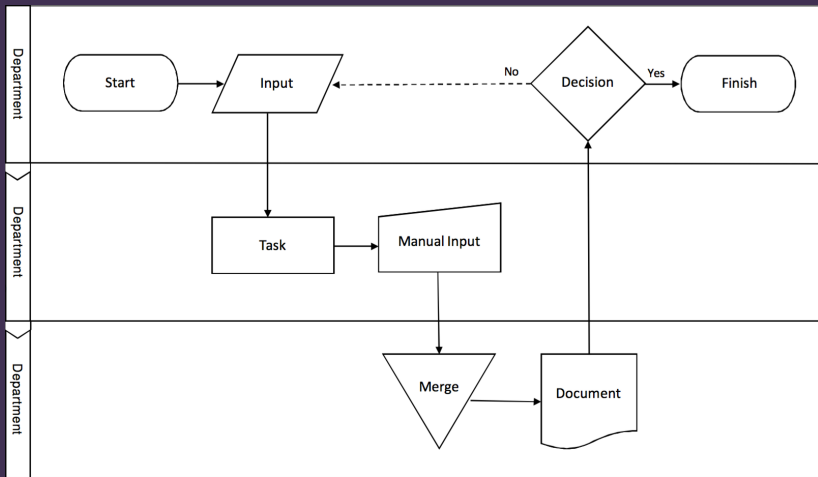


Used to connect one part of  
a flowchart to another.



Connector used to connect one  
page of a flowchart to another.

# Swimlanes



# Activity

- ▶ In **groups of 2-3**
- ▶ **Draw** a flowchart for **logging into Facebook**
- ▶ Draw your flowchart using **pen and paper**
- ▶ Include at least two swimlanes: **the user's browser/device** and **the Facebook server**
- ▶ Take a **photo** of your flowchart and post it on **Slack**

# Software for drawing flowcharts

# Software for drawing flowcharts

Intended for drawing flowcharts:

- ▶ Gliffy <https://www.gliffy.com>
- ▶ LucidChart
- ▶ Microsoft Visio

# Software for drawing flowcharts

Intended for drawing flowcharts:

- ▶ Gliffy <https://www.gliffy.com>
- ▶ LucidChart
- ▶ Microsoft Visio

Can draw flowcharts:

- ▶ Microsoft PowerPoint
- ▶ Google Docs

# Software for drawing flowcharts

Intended for drawing flowcharts:

- ▶ Gliffy <https://www.gliffy.com>
- ▶ LucidChart
- ▶ Microsoft Visio

Can draw flowcharts:

- ▶ Microsoft PowerPoint
- ▶ Google Docs

If you're desperate:

- ▶ Any drawing package (Inkscape, Adobe Illustrator, Apple Keynote, ...)
- ▶ MS Paint



# Pseudocode



# Pseudocode

# Pseudocode

Flowcharts are useful, but...

# Pseudocode

Flowcharts are useful, but...

- ▶ Can be time-consuming to draw

# Pseudocode

Flowcharts are useful, but...

- ▶ Can be time-consuming to draw
- ▶ Do not reflect structured programming concepts well

# Pseudocode

Flowcharts are useful, but...

- ▶ Can be time-consuming to draw
- ▶ Do not reflect structured programming concepts well

**Pseudocode** expresses an algorithm in a way that looks more like a structured program

# Pseudocode example

```
print "How old are you?"  
read age  
if age < 13 then  
    print "You are a child"  
else if age < 18 then  
    print "You are a teenager"  
else  
    print "You are an adult"  
end if
```

# Pseudocode example

```
sum  $\leftarrow$  0                                ▷ initialisation  
for i in 1, ..., 9 do  
    sum  $\leftarrow$  sum + i  
end for  
print sum                                ▷ print the result
```



# Pseudocode example

```
 $a \leftarrow 1$                                 ▷ initialisation  
while  $a < 100$  do  
     $a \leftarrow a \times 2$   
end while  
print  $a$                                 ▷ print the result
```

# Formatting pseudocode

# Formatting pseudocode

- ▶ Pseudocode is a **communication tool**, not a **programming language**

# Formatting pseudocode

- ▶ Pseudocode is a **communication tool**, not a **programming language**
- ▶ Important: **clear, concise, unambiguous, consistent**

# Formatting pseudocode

- ▶ Pseudocode is a **communication tool**, not a **programming language**
- ▶ Important: **clear, concise, unambiguous, consistent**
- ▶ **Not** important: adhering to a strict set of style guidelines, ensuring direct translatability to your chosen programming language

# Level of abstraction

# Level of abstraction

Whether working with flowcharts or pseudocode, choose your **level of abstraction** carefully

# Level of abstraction: Good

Fill kettle

Turn kettle on

Put instant coffee in mug

**if** sugar wanted **then**

    Add sugar

**end if**

Wait for kettle to boil

**if** milk wanted **then**

    Pour water to  $\frac{4}{5}$  full

    Add milk

**else**

    Fill mug with water

**end if**

Stir



# Level of abstraction: Not so good

Position kettle beneath tap

Turn tap on

**while** water is below halfway point **do**

    Wait

**end while**

Turn tap off

Place kettle on base

Press power button

...

# Level of abstraction: Silly

Place right palm on kettle handle

Bend fingers on right hand

Lift arm upwards

**while** tap spout is not directly above kettle **do**

    Move arm to the right

**end while**

Place left palm on tap handle

Bend fingers on left hand

Rotate left hand

...

# Level of abstraction: also silly

Make a cup of coffee

# Activity

A number guessing game: The computer chooses a number between 1 and 20 at random. The player guesses a number. The computer says whether the guessed number is “too high”, “too low” or “correct”. The game ends when the correct number is guessed, or after 5 incorrect guesses.

- ▶ In **groups of 2-3**
- ▶ **Write** pseudocode for the number guessing game
- ▶ **Post** your pseudocode on Slack

# Activity

- ▶ In **groups of 2-3**
- ▶ **Choose** an algorithm from one of the following:
  - ▶ Lego Robot Olympiad
  - ▶ COMP120 Tinkering Graphics
  - ▶ COMP150 Game Development Project
- ▶ **Express** the algorithm as a flowchart **and**
- ▶ **Express** the algorithm as pseudocode
- ▶ **Post** both your flowchart and your pseudocode on Slack

# Worksheet B

- ▶ Flowcharts and pseudocode
- ▶ Due in class **next Tuesday**