COMP110: Principles of Computing

3: Basic data types

FALMOUTH UNIVERSITY

Research journal

Research journal

- **Read** some seminal papers in computing (listed on the assignment brief)
- **Choose** one of them
- **Research** how this paper has influenced the field of computing
- **Present** (briefly!) your findings
- **Write up** your findings
  - Maximum 1500 words
  - With reference to appropriate academic sources

Marking rubric

See assignment brief on LearningSpace

Timeline

- **Presentations** in week 7 (4th and 8th November)
- **Peer review** in week 8 (14th November)
- **Deadline** shortly after (check MyFalmouth)
- Finding and reading academic papers takes time and effort — don't leave it until the last minute!

Worksheets

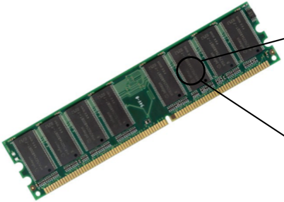
- Worksheet 2 (Binary numbers) due **tomorrow**
- Worksheet 3 (Flowcharts and pseudocode) due **next Friday**

Data types

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

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

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

Numeric types

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

Integers as bytes

- A **32-bit** integer is stored as a sequence of **4 bytes**
- Example: 314159 in decimal = 100 11001011 00101111 in binary
- Stored as four bytes:

  00000000 00000100 11001011 00101111

  or in hexadecimal:

  00 04 *CB* *2F*

- Similarly for other sizes of integer: an n-bit integer is stored as $n \div 8$ bytes
- You can think of this as a base-256 numbering system

Endianness

- Integers are stored either **big endian** or **little endian**
- Big endian: the **most significant byte** comes first

00 04 *CB* *2F*

- Little endian: the **least significant byte** comes first

2F *CB* 04 00

- Modern PCs (Intel x86 based) use little endian
- Little endian may seem unintuitive
- However it is more efficient when programs need to convert one size of integer to another

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

- `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 (completely different sequences of bytes)
  - However `==` etc still know how to compare them sensibly

Other number formats

- **Fixed point**: alternative format for non-integer numbers
  - More on this later
  - E.g. `decimal` module in Python
- **Rational numbers**: store fractions as numerator and denominator
  - E.g. `fractions` module in Python
- **Complex numbers**: stored as a pair of floating point numbers for real and imaginary parts
  - E.g. `complex` type in Python

String types

Strings

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

String representation

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

What is a character?

- Broadly speaking, a single **printable symbol**
- There are also some special **non-printable characters** e.g. line break

ASCII

- American Standard Code for Information Interchange
- Defines a standard set of 128 characters (7 bits per character)
- Originally developed in the 1960s for teletype machines, but survives in computing to this day
- 95 printable characters: upper and lower case English alphabet, digits, punctuation
- 33 non-printable characters

Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value
00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

ASCII

- ASCII works OK for English
- Standards exist to add another 128 characters (taking us to 8 bits per character)
- E.g. accented characters for European languages, other Western alphabets e.g. Greek, Cyrillic, mathematical symbols
- However 256 characters isn't enough...

Unicode

- Standard character set developed from 1987 to present day
- Currently defines 137994 characters (Unicode 12.1)
- First 128 characters are the same as ASCII
- Covers most of the world's writing systems
- Also covers mathematical symbols and emoji

Encoding Unicode

- **UTF-32** encodes characters as 32-bit integers
- **UTF-8** encodes characters as 8, 16, 24 or 32-bit integers
  - 8-bit characters correspond to the first 128 ASCII characters $\implies$ backwards compatible
  - More common Unicode characters are smaller $\implies$ more efficient than UTF-32

String representation

- `"Hello world!"` in ASCII or UTF-8 encoding:

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

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]
- `"Haha 😂"` encoded in UTF-8:

H	a	h	a	space	😂				null
72	97	104	97	32	240	159	152	130	0

Strings in Python

- Python 2 had separate types for ASCII and Unicode strings: `str` and `unicode`
- Python 3 has just the `str` type, which uses Unicode
- String literals are wrapped in `'single quotes'` or `"double quotes"` (there is no difference)

Escape sequences

- Backslash \ has a special meaning in string literals — it denotes the start of an **escape sequence**
- Typically used to write **non-printable characters**
- Most useful: `"\n"` is a new line
- How to type a backslash character? Use `"\\"`

String literal tricks in Python

- Use triple quotes `'''` or `"""` for a multi-line string
- Use `r" "` or `r' '` to turn off escape characters (useful for strings with lots of backslashes, e.g. Windows file paths, regular expressions)

Text files

- Stored on disk as essentially one long string
- Line endings are denoted by non-printable characters
  - Unix format: line feed character (ASCII/UTF-8 character 10, `"\n"`)
  - Windows format: carriage return character (ASCII/UTF-8 character 13) followed by line feed, `"\r\n"`
  - Most text editors can handle and convert both formats
  - Most languages allow files to be opened in “text mode” which automatically converts

Other types

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

- 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

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

Checking types in Python

- Call `type()` to check the type of a variable or value
- Note that `type()` returns a value of type `type`
- You can use these `type` values like any other value, e.g.

```
if type(x) == int:
    print("x has type int")
elif type(x) == type(y):
    print("x and y have the same type")
```

Other types

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

FALMOUTH
UNIVERSITY

Converting types

Weak vs strong typing

- In **weakly typed** languages, a variable can hold a value of any type
  - Examples: Python, JavaScript
- In **strongly typed** languages, the type of a variable must be **declared**
  - Examples: C#, C++, Java

Weak typing (example in Python)

```
x = 7
# Now x has type int

x = "hello"
# Now x has type string
```

Strong typing (example in C#)

```
int x = 7;
// x is declared with type int

x = "hello";
// Compile error: cannot convert type "string" to "int"
```

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

Operations on types

- Certain operations can only be done on certain types of values
- Can add two ints: `2 + 3` → `5`
- Can add int and float: `2 + 3.1` → `5.1`
- Can add two strings: `"COMP" + "110"` → `"COMP110"`
- Can't add string and int: `"COMP" + 110` → error

Implicit type conversion

- The type casts we saw a few slides ago are **explicit**
- Some languages (not Python) can perform **implicit** type casts to make operations work
- Sometimes called **type coercion**
- E.g. in JavaScript, `"COMP" + 110` → `"COMP110"`
- The integer `110` is implicitly converted to a string `"110"` to make the addition work
- Equivalent in Python with explicit casts:

```
"COMP" + str(110)
```

Dangers of implicit type conversion

- Rules for implicit type conversion can sometimes be confusing
- E.g. in JavaScript:
  - `"5" + 3` → `"53"`
  - `"5" - 3` → `2`