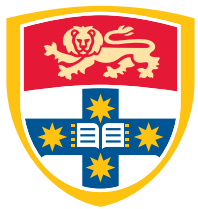
Introduction to Programming

Dr. John Stavrakakis

School of Computer Science, University of Sydney



Copyright Warning

Week 2: Variables, Expressions

We will cover: Variables (numbers, characters and Strings), mathematical and Boolean (true/false) expressions

You should read: Sections 1.2 of Sedgewick

Lecture 3: Variables and Expressions

Storing and manipulating information in memory

Binary Numbers

It helps to know about binary numbers in your programming, because everything in a computer is stored in binary digits (bits).

You can think of Boolean[1] values as *binary* numbers (in a sense)

$$TRUE = 1, FALSE = 0$$

[1] Why have I written "Boolean" here and not `boolean`? Because a Boolean variable is named after George Boole, who developed the concept of what we now call Boolean algebra in 1854. The variable type in Java uses the lower-case 'b' because it's a *primitive type*, and the naming convention for primitives is to use a lower case.

Integers are stored as binary

All integers can be thought of as binary numbers: 0, 1, 10, 11, 100, 101, . . . :

base 10	**base 2 / binary**	**expansion**
0	0	0
1	1	1×2^0
2	10	$1 \times 2^1 + 0 \times 2^0$
3	11	$2^1 + 2^0$
5	101	$2^2 + 2^0$

Bytes

```
00000000 = 0
00000001 = 1
00000010 = 2
00000100 = 4
00001000 = 8
00010000 = 16
00100000 = 32
01000000 = 64
10000000 = 128
--------
76543210 bits
```

In fact everything in your computer is stored as bits.

A byte is eight *bits*.

Bit means “binary digit”.

Bits

1s and 0s are easy to store as “on” and “off” so binary has become the universal way to store all electronic information.

If the bits are

$$d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0$$

then the total value is

$$d_7 2^7 + d_6 2^6 + d_5 2^5 + d_4 2^4 + d_3 2^3 + d_2 2^2 + d_1 2^1 + d_0 2^0$$

which is equal to

$$128 d_7 + 64 d_6 + 32 d_5 + 16 d_4 + 8 d_3 + 4 d_2 + 2 d_1 + d_0$$

E.g.,

$$01101010 = 0 + 64 + 32 + 0 + 8 + 0 + 2 + 0 = 106$$

Hexadecimal

Sometimes it's handy to have a more compact way to store numbers in a range that's a power of two.
Noting that 2^4 is 16, we use the first 6 letters of the alphabet to represent 10, 11, . . . , 15.

This is more compact: a byte just takes 2 characters: `00` (0) to `FF` (255).

To indicate that a number is to be interpreted in this way we put `0x` in front: `0x00` and `0xFF`.
Two-byte numbers need four such characters, so look like this: `0x0000` to `0xFFFF` (65535).

binary	hexadecimal	decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Variables

Programs would be pretty simple and useless if they couldn't handle data that could take different values. What if we wanted to print something other than "Hello World!" or "I'm sorry Dave, I can't let you do that" ?

```
1 print("You bought 4 items")
2 print("It costs you $12.50")
3 print("Served by autobot 526 on 24/02/2016")
```

To do this we need to have some *variables*.

Variables (cont.)

A variable is a piece of information that's stored in memory so we can access it by giving its name later, like this:

```
1 x = 4
2 y = 8
3 z = x + y
```

- Here any time we need to get the value of x we just call it by name.
- The name doesn't matter: it's just a label.

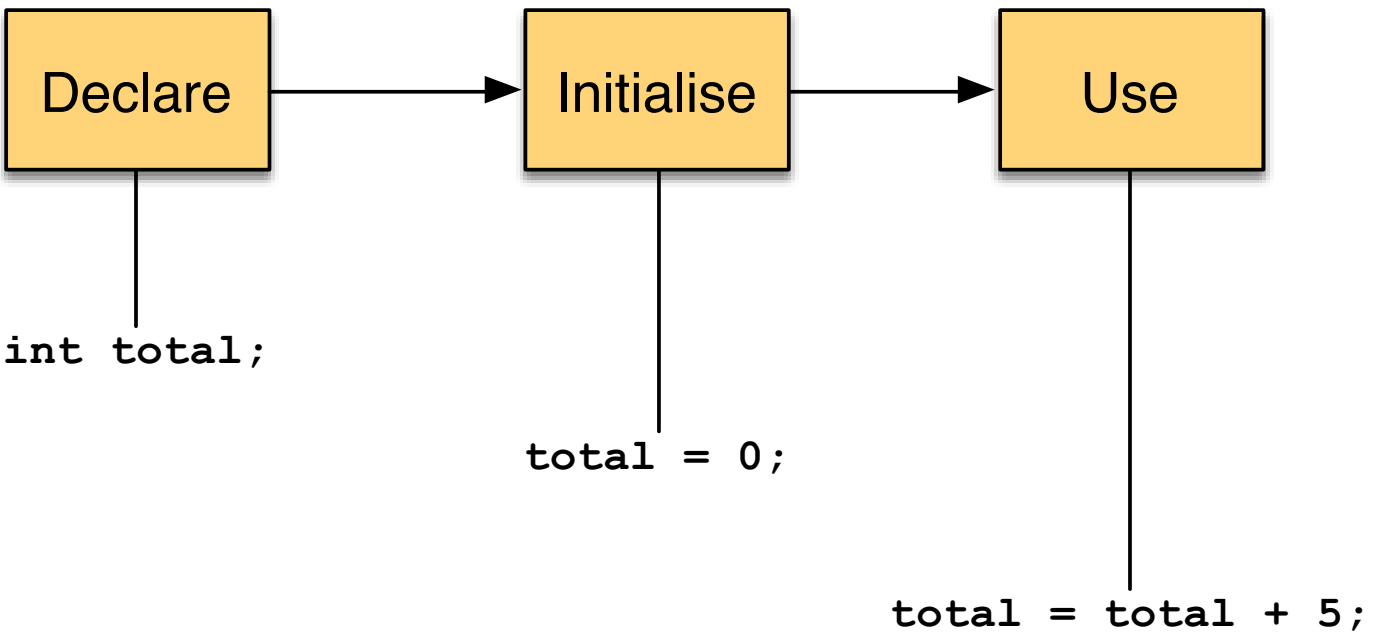
Variables (cont.)

Primitive types are the simplest type of variables e.g. an integer (int). They use a fixed amount of memory. Simple math operations e.g. + - * / % can use the value stored in that variable, or store the result...but in Python, everything is an *Object*!

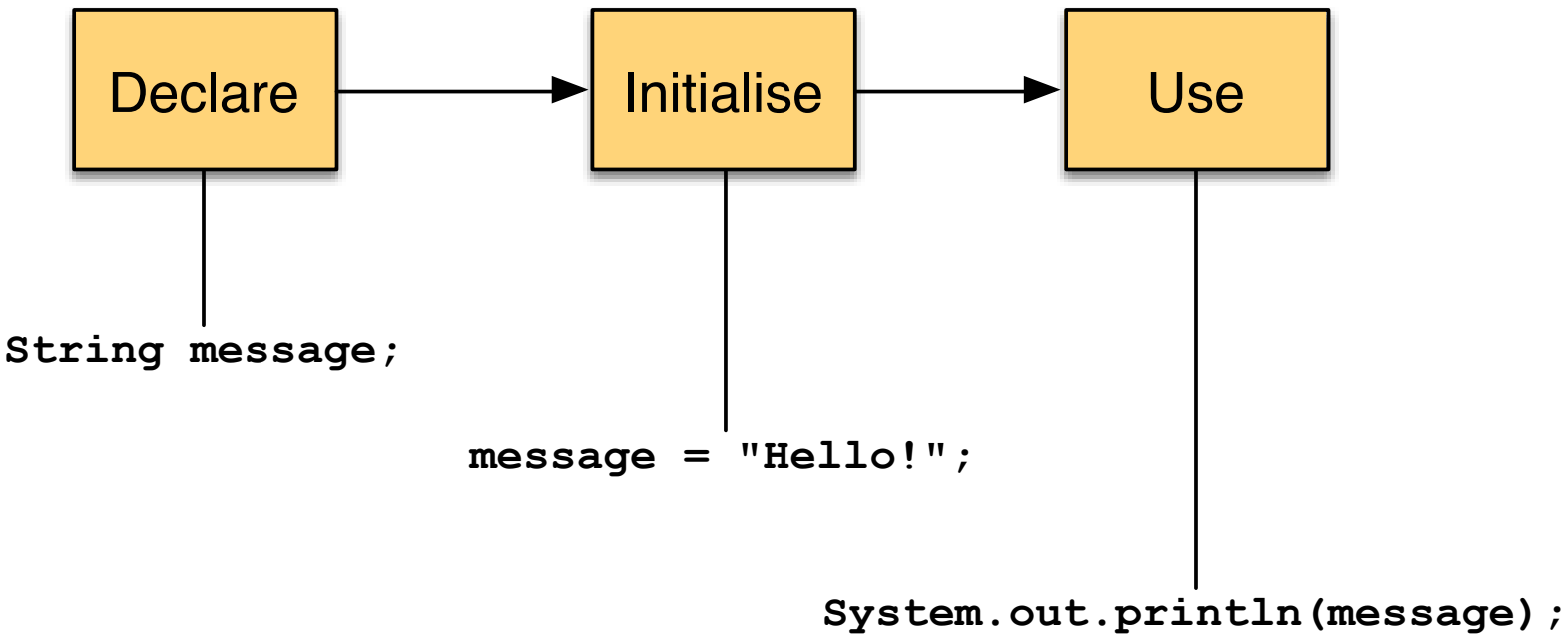
Objects are blocks of memory that contain *related* data and associated operations for that data. An integer is represented within an object as a sequence of bits and we can perform operations such as addition, subtraction etc.

One object can contain many other Objects. The amount of memory used by an Object can change over time. They can have customised operators *defined by the programmer*. Objects will be discussed in later lectures.

Declare → Initialise → Use



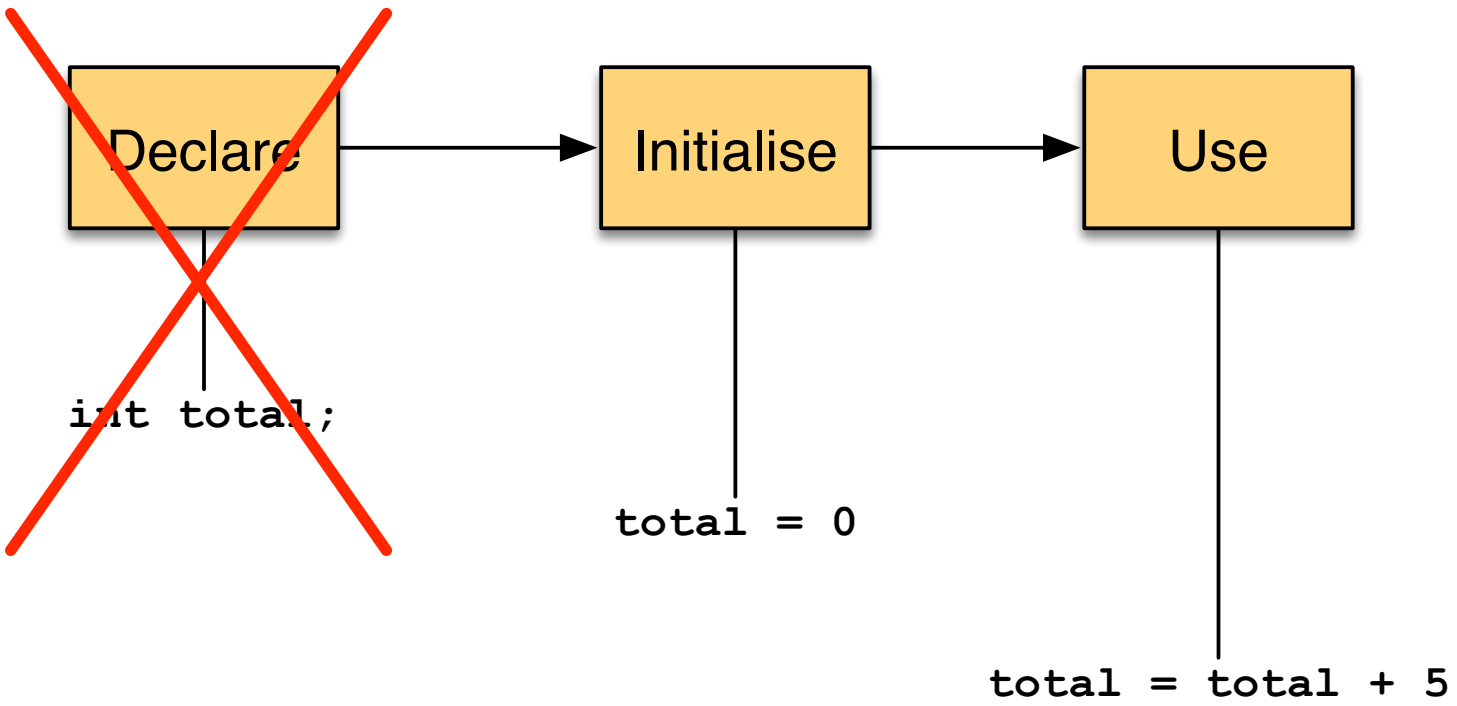
Declare → Initialise → Use



You have to have things in this order, else there will be . . . *trouble*.

Declare → Initialise → Use

Python declare and initialise in one step.



It is strongly encouraged that you initialise every variable. Life will be easier.

Variables must *declared* before they're used

So you have to *declare*/*initialise* a variable before you use it: you have to decide what type you need first, like an integer, character or string, before you start using it.
Code that misses the declaration out or gets it in the wrong order, like this:

```
1 print("the temperature is " + temperature)
2 temperature = 26
```

Simply won't *compile*:

```
> python DeclareVariables.py
Traceback (most recent call last):
  File "DeclareVariables.py", line 1, in <module>
    print("the temperature is " + temperature)
NameError: name 'temperature' is not defined
```

Initialising Variables

Variables have to be *initialised*.

> Initialising is giving the very first value to a variable.

Good programmers *always* initialise their variables to something useful.

In Python, almost all variables are initialised when they are introduced.

```
1 num = get_user_guess()
2 # we have to declare msg, so I just use any value
3 msg = 'ponyrides in the wilderness'
4 if num == 34:
5     msg = 'correct'
6 print(msg)
```

Initialising Variables (cont.)

```
1 num = get_user_guess()
2 msg = 'incorrect'
3 if num == 34:
4     msg = 'correct'
5 print(msg)
```

```
1 num = get_user_guess()
2 if num == 34:
3     msg = 'correct'
4 else:
5     msg = 'incorrect'
6 print(msg)
```

Scope: Variables have a lifetime

In the code you've seen so far the variables have been short-lived and this hasn't mattered one bit.

In the programs we are writing now, without classes or functions, the variables introduced will live until the end of the program.

```
1   x = 5
2   if True:
3       print(x)
4       y = 7
5       if True:
6           print(x)
7           print(y)
8           z = 3
9   print (x)
10  print (y)
11  print (z)
```

This is a very important concept for you to remember because they have to be tracked. Later in the course we will revisit the idea of scope.

What kinds of variables are there?

Python provides builtin *types* to hold information for numbers and text.
Python 2

Name	Kind	Memory	Range
`bool`	boolean	1 byte	True or False
`int`	integer	4 bytes	$[-2147483638, 2147483647]$
`float`	floating-point	8 bytes	$\approx \pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$
`str`	string	value dependent	$[0, \mathit{verylong}]$

What kinds of variables are there? (cont.)

Python 3 (this course)

Name	Kind	Memory	Range
`bool`	boolean	1 byte	True or False
`int`	integer	unknown	no limit
`float`	floating-point	maybe 8 bytes	depends on platform
`str`	string	value dependent	$[0, \textit{verylong}]$

`boolean` variables

A Boolean variable, in Python, is assigned a value *True* or *False*.

You will come across Boolean variables *very* commonly, and you will mainly use them to test whether certain things are true or not.

boolean code example

```
1 foo = True
2 bar = False
3 flag = foo
4 flag = foo and bar
5 print("flag = " + str(flag))
```

The above shows how you can *initialise* and *operate* on a `boolean` variable called `flag`.

Notes:

- You use the values `true` and `false` directly;
- You can assign the value of one boolean variable to another;
- You can calculate the result of two `boolean` values with & operator.

int example

```
1 a = 45
2 b = 23
3 c = a + b
4 print( str(a) + " + " + str(b) + " = " + str(c) )
```

Compiling and running the above we get

```
~> python IntTest.py
45 + 23 = 68
```

Note how I had to separate the parts of the equation that I wanted to print out. If I had put just this:

```
1       print( str(a + b) + " = " + str(c) )
```

then I would have seen the output

```
      68 = 68
```

. . . which isn't as interesting.

`float` variables

To represent numbers using a decimal point e.g. 3.14, 0.0023, -6.21×10^3

`float` is referred to as *floating point* types.

Typical `float` type uses 4 bytes of memory, but Python 3+ has arbitrary precision.

The range and the accuracy of floating point types depends on how many bits are used in the *coefficient*, and the *exponent*. Python will manage this automatically.

`float` example

```
1 radius = 0.51238761523
2 area = radius * radius * 3.141592659
3 print("The area of a circle of radius "
4                 + str(radius) + " is " + str(area) )
```

Compiling and running the above we get this:

```
~> python FloatTest.py
The area of a circle of radius 0.51238761523 is 0.8247970926722155
```

`float` example (cont.)

Note that I used the multiplication sign '`*`', rather than something like "`^2`", which *doesn't mean squared*. There is a symbol for power, it is `**` e.g. `x ** 2`.

To square a number, it's simplest to multiply it by itself explicitly.

To raise a number to a power, use this:

```
1        radius = 0.4
2        volume = pow(radius, 3.0) * 4.0 / 3.0 * 3.1415926
```

You can also get a more accurate from Python itself than trying to remember π:

```
1  import math
2  radius = 0.4
3  volume = pow(radius, 3.0) * 4.0 / 3.4 * math.pi
4  print("The volume of the sphere is " + volume)
```

When to use integer or floating point

First identify the information that is going to be stored and how that changes over time.

Are they whole numbers or can there be a fractional component?

accuracy if you are only dealing with integer values, integers are exact but floating point numbers are not

range If you're dealing with very large or very small numbers, you have to use floating point types: their range is much bigger

In this course, you should pick either `int` or `float`.

Is there any danger of using a floating point type in your program if only integers are expected?

A character

A character is an human defined symbol. The letter `'A'`, the letter π, a punctuation mark ;

Each character is in fact a graphic. It is stored in the computer.

Displaying the character requires identification among many characters. A number is used.

A character set is a mapping from a numbers to characters.

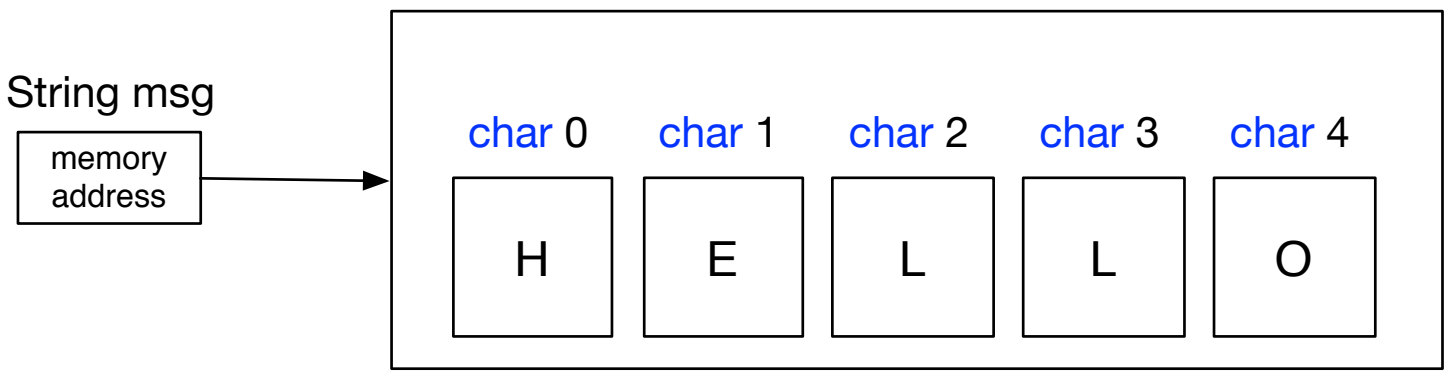
The string type

A string datatype is a sequence of characters *stringed* together to represent text information.

a `string` is an object

- it can have a few characters or several thousand.
- it has special operators to query or manipulate the text information.

```
1       msg = "HELLO"
```



The string operators (str)

+ *operator*. Creates a new string from the concatenation of two existing strings.

len() method - returns an integer. The number of characters in the string.

== *operator* - returns a boolean value. `True` if this string matches another, `False` otherwise.

many many more!

string example

```
1  word1 = "Hello"
2  word2 = "World!"
3  twoWords = word1 + word2
4  print(twoWords)
5
6  word1len = len(word1)
7  print("string word1 has " + str(word1len) + " characters.")
8  print("string twoWords has " + str(len(twoWords)) + " characters.
       ")
9
10 isequal = (word1 == word2)
11 print("Does string word1 match string word2 ? "
12                + str(isequal) )
```

The above, when compiled and run, gives

```
HelloWorld!
The string word1 has 5 characters.
The string twoWords has 11 characters.
Does string word1 match text of word2 ? false
```

string notes

string looks more complicated than `int` and `float`. There is internal data and many operators.

The *methods* are *called* and used to query the string object for information.

string is a good example of a *reference type*, which is a very important part of object-oriented programming that we will learn a great deal more about. Because strings are so useful, we will begin using them now.

`None` then it means it has no memory associated with it — and you really shouldn't use it or very bad things will happen!