Introduction to Programming

Dr. John Stavrakakis

School of Computer Science, University of Sydney

Copyright Warning

Lecture 4: Expressions, Command-line arguments, Conventions, and Pseudocode

Manipulating information, Immediate input to your program; Style; how to write algorithms

Using the command-line

Getting information in to your program

Reading values in with input

Recall that we can use a input to read in values to make an interactive program:

```python
print("Enter your height in metres: ")
input_variable = input()
height = float(input_variable) # ASSIGNING the variable
print("You entered " + str(height) + "m.")
print("If you were 10% bigger you'd be "
          + str(height*1.1) + "m.")
```

```
~> python ReadADouble.py
Enter your height in metres:
1.82
You entered 1.82m.
If you were 10% bigger you’d be 2.0020000000000002m.
~>
```

Using the command-line arguments

If you don't want to use `input`, such as if you want to run your program like this:

```
~> python GrowMe.py 1.82
You entered a height of 1.82m.
If you grow by 10% then you'll be 2.002m.
```

…then you have to use the *command-line arguments*

The command-line arguments

When you run a python program from the command-line or terminal you do something like this:

```
> python HelloWorld.py
Hello, World!
```

…but you can also give a program information directly, as soon as you call it, like this:

```
> python Sum.py 4 5.5
The sum of 4 and 5.5 is 9.5
```

How does this work?

The `main` function

Different computer languages have different main function. The main function is supposed to be the entry point to the program, where the first computer instruction begins.

Python's does not have a `main` function. Instead, the code always begins from the beginning of the .py file.

In the spirit of tradition, programmers may have something like this to allow other readers to see where they start doing useful things[1]

```
if __name__ == "__main__":
   print("Hello World!")
```

[1] Technically, there can be anything that executes before this. The programmer is not changing the entry point of the program, just directing attention

Program arguments

Now that we know the starting point, we can obtain the values we provide at command line

```
    import sys
    print(sys.argv[0]) # print program name
    print(sys.argv[1]) # print 1st argument
    print(sys.argv[2]) # print 2nd argument
```

The variable `sys.argv` gives you a way to access the pieces of information you provide, like 4 and 5.5 above.

Program arguments (cont.)

The way we do that is by referring to them using the square brackets, also called the “index operator”, like this:

```python
import sys

x = float(sys.argv[1])
y = float(sys.argv[2])

print("The sum of " + str(x) + " and " + str(y)
    + " is " + str(x+y));
```

The arguments are always `Strings`: they have to be converted into numbers if you want to use them as numbers.

float()

`float()` is a way of converting a `String` into a `float`.

This is extremely useful if you need to read in floating-point numbers to a program when you run it.

Use it like this:

```
        x = float("12.3")
```

now the String “12.3” has been *parsed* — that is, *read and understood* — as a `float` value, and then stored in `x`.
The original `String` is left untouched: it doesn’t change.

(There is also `int()` : it parses a string like “123” as the integer number 123.)

Expressions

How to do calculations

Simple expressions

An *expression* is just a combination of variables and other items that can be *evaluated*, and will have a *value* like `True`, `False`, 25, 1.4142, etc.

Here are some expressions:

4	A simple number;
$x + y$	A mathematical formula;
$\sqrt{3}$	Another one;
$(A == B)$	A Boolean expression, which will evaluate to either true if A equals B, or false otherwise

Assignment

We *assign* values to variables using a single equals sign, like this:

```
x = 4
y = x
```

After this, y has the value 4.

```
x = 2*y
```

Now x has the value 8, and y still has the value 4.

We construct expressions using *operators*

An operator is a symbol or small set of symbols that let you perform some kind of operation on the *operands*.
Here are some:

`x = 5`	*assigns* the value of `x` to be 5
`a + b`	*adds* `a` and `b`
`!sad`	*negates* the Boolean value of the variable `sad`
`x++`	most programming langauges: adds 1 to the variable `x`: this is called *incrementing*. In Python, this is equivalent of `+(+x)`, where `+x` has no effect.
`a < 23`	*compares* the value of `a` to 23

Operator terminology

Operators work on *operands*. They may or may not modify the operand.

An operator can be

unary	operating on one operand;
binary	operating on two operands
ternary	operating on three operands

Assignment operator =

There are several operators in Python to make your life easier. The most common you'll probably use is the *assignment* operator:

$$=$$

We've seen this before: use = to assign the value on the left to take the value on the right:

$$lvalue \leftarrow rvalue$$

$$lvalue = rvalue$$

And remember the equality operator is ==: it is used to compare whether two *primitive types* are equal.

Operators +, -, *, /

The next set of operators are very straightforward: they are the standard *operators* of mathematics:

$+$	$-$	$\times$	$\div$	(mathematical)
`+`	`-`	`*`	`/`	(code)

For example:

```
x = 5
y = 3
z = x + y  # z gets the value of (x+y), which is 8
x = -x         # now x is -5
z = z + x    # now z is 3
y = y * 2     # now y is 6
```

Warnings about operators

Integer division

```
x = 3 / 2
print(x)

y = 1 / 0
print(y)
```

Embedding shorthand operators is dangerous.
What do you think happens in the following?

```
x = 4
matches = (++x == 3)
y = (x + 7) + ( --x )
```

This is how bugs get introduced. This needs to be readable. It should be expanded to show each calculation in a separate statement.

```
x = 4
y = 15
```

Simple Calculations

```
x = 6
y = ++x
z = (x + --y)
n = 0
n += 1
print("x = " + str(x))
print("y = " + str(y))
print("z = " + str(z))
print("n = " + str(n))
x = 5
r = 1.2
s = x * r
print("x = " + str(x))
print("r = " + str(r))
print("s = " + str(s))
s = s - 1.1
print("s = " + str(s))
```

What does this print?

Simple Calculations

```
~> python SimpleOperators.py
x = 6
y = 6
z = 12
n = 1
x = 5
r = 1.2
s = 6.0
s = 4.9
```

+ and Strings

Above, in the lines

```
x = 5
r = 1.2
s = x * r
```

and

```
print("r = " + str(r))
```

there is a “+” sign in arguments of the `print` method call. It’s there to *concatenate* two `Strings`.

`"x = "` is a `String`, and `print` can only print `String` objects, so it converts the whole expression to a single `String`.

+ and Strings (cont.)

What do you think this prints out:

```
print("5" + "3")
print(5 + 3)
```

???

In general you can use the "+" to concatenate any two `Strings`, e.g., like this:

```
    msg = "Hello, " + sys.argv[1] + ", how are you?"
```

Operators +=, -=, *= and /=

These operators are a very nice shorthand. They all operate in the same way, by modifying the operand on the left, using the operand on the right.

shorthand	**equivalent to**
`x += n`	`x = x + n`
`x -= n`	`x = x - n`
`x *= k`	`x = x*k`
`x /= k`	`x = x/k`

In general,

```
x □= y
```

is equivalent to

```
x = x □ y,
```

for whatever □ is.

Equality operator ==

This binary operator should be very familiar by now: use == to return the value *true* when the two operands are equal:

left value == right value

is true if and only if the two values are the same.

Comparing ints and booleans

```python
x = 4
y = 4
z = 2
xySame = (x == y)
xzSame = (x == z)
print("xySame = " + str(xySame))
print("xzSame = " + str(xzSame))
print("(xySame == xzSame) = " + str(xySame ==  xzSame))
```

```
> python Equality.py
xySame = True
xzSame = False
(xySame == xzSame) = False
```

Comparing `Strings`: *msg1 == msg2* ?

Yes. The == method will visit the area of memory of the two strings and compare the content.

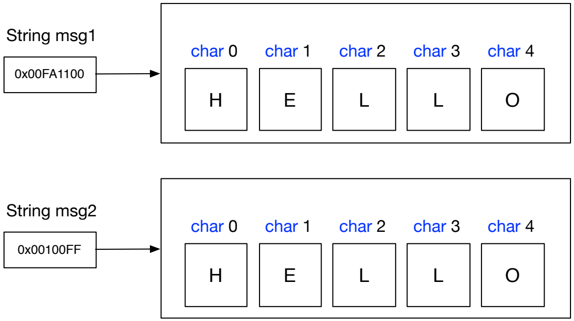
```python
msg1 = "hello"
msg2 = "hello"
if msg1 == msg2:
        # ...
```

Python will generalise the == to apply to other kinds of objects

Comparing Strings: *msg1* `is` *msg2* ?

Don't use `is` to compare String content

`is` compares two single values. The value of the String variable is a *memory address*. This has no information about the content of that area of memory.



not!

There are two ways to negate. There is the keyword `not` and the negation operator: “`!`” — it’s the exclamation mark. In code you’ll see this quite often, for example in expressions like

```
x = 1
y = 2
same = x != y
print(same)
```

which is true if `x` is *not* equal to `y`, or like this:

```
x = 1
y = 2
same = not ( x == y )
print(same)
```

which is true if the statement “`x == y`” is *False*.

In mathematical symbols we use the “negate” symbol $\neg$ for “not”.

- The value of $\neg x_1$ is true if, and only if, x_1 is *False*.

Parentheses

I've been using expressions inside parentheses `(`, `)`.

Expressions, when they are evaluated/executed, have a *value*.

That means if I write something like

```
x = 4
y = 4
```

then if the expression `(x == y)` is executed, it will give the result "`True`".

Writing the expression in parentheses is usually a good idea. It avoids confusion!

```
not (x == y)
```

is equivalent to

```
(x != y)
```

but

```
(not x) == y is different
```

And and Or

Let's think about a set of variables, called $x_1, \dots x_k$.

- The value of $(x_1$ AND $x_2)$ is true if, and only if, both x_1 and x_2 are true.
- The value of $(x_1$ AND x_2 AND ... AND $x_k)$ is true if, and only if, all of the x_i are true.
- The value of $(x_1$ OR $x_2)$ is true if, and only if, at least one of x_1 and x_2 is true.
- The value of $(x_1$ OR x_2 OR ... OR $x_k)$ is true if, and only if, at least one of the x_i is true.

In Python we write `and` for logical AND, and `or` for logical OR. [2]

[2] It is not obvious, but other languages will use other symbols

Operator Precedence

What is the result of:

```
x = 1 + 5 * 3 + 2
y = 8 / 4 - 1 / 1 - 1
```

This particular programming language has this order: Brackets, Operators, Division / Multiplication, Addition / Subtraction (BODMAS)

Always *use parentheses* to be clear.

```
x = (1 + 5) * (3 + 2)
y = 8 / (4 - (1 / 1) - 1)
```

Assignment

In Python, if we want to set the value of something we use *assignment* that looks like this:

```
x = -1/2
```

Just remember the value on the *left* gets the value of the expression on the *right*.
In pseudocode we'd write $x \leftarrow \frac{-1}{2}$: the left-arrow is often called "gets" to mean the variable on the left *gets* the value on the right.

Assignment is not equality

Saying $x = 3$ in mathematics means “x is a variable whose value is currently 3”.
In that sense the equivalent statement is $3 = x$, but writing that in a Python program doesn’t make sense: you would be attempting to change the value of 3!

If you want to test whether x has the value 3, you would evaluate the following expression:

```
(x == 3)
```

which has the Boolean value *True* if x really does equal 3 and the value *False* otherwise.

Don’t confuse ‘=’ (assignment) with ‘==’ (equality comparison)!

Comparison — warning!

***Comparing floating point numbers for equality is a BAD IDEA*.**

floating point numbers are stored to finite precision. $\frac{1}{3}$ is not stored exactly, but approximated.

```python
f1 = 0.00015 + 0.00015
f2 = 0.0002 + 0.0001
matches = ( (f1 == f2) )
print("f1 = " + str(f1) )
print("f2 = " + str(f2) )
print("matches = " + str(matches))
```

prints out

```
f1 = 0.0003
f2 = 0.00030000000000000003
matches = False
```

This is a serious problem.

Recall

Do you understand what each step of this program is doing?

```python
print("Enter your height in metres: ")
input_variable = input()
height = float(input_variable) # ASSIGNING the variable
print("You entered " + str(height) + "m.")
print("If you were 10% bigger you'd be "
           + str(height*1.1) + "m.")
```

```
~> python ReadADouble.py
Enter your height in metres:
1.82
You entered 1.82m.
If you were 10% bigger you'd be 2.0020000000000002m.
~>
```