Fundamentals of Programming 1

Lecture 3

Aurelia Power, ITB, 2018

Arithmetic Expressions and Operator Precedence

- An expression is a combination of symbols (operators) and variables/values (operands) that evaluate to a certain value.

- **EXAMPLES**:
  3 + 3 * 2
  n / 2 – 12 * x (assume that n and x have associated values)

- An expression may be further broken down into sub-expressions:
  3 + 3 * 2 where 3 * 2 is a sub-expression that evaluates to 6, then 3 + 6 operation is carried out; so the result of the entire expression is 9
- Some arithmetic operators are evaluated before others: they have precedence over others.
- In addition, some operators are evaluated from left to right, whereas others are evaluated right to left.

Arithmetic Expressions and Operator Precedence

Precedence level	Operator(s)	Associativity
First	** (exponentiation)	Right – to – left
Second	- (negation)	Left – to – right
Third	* (multiplication), / (division), // (floor division), % (modulus)	Left – to – right
Fourth	+ (addition), - (subtraction)	Left – to – right

- **EXERCISE:** evaluate the following expressions

  4 / 2 ** 2 + 1 // 2 1.0

  - 3 / 1 * 2 ** 0 + 2 -1.0

  10.5 // 1 * 3 30.0

Arithmetic Expressions and Operator Precedence

- Sometimes, we need lower level precedence operations to be performed before higher ones;
- For example, in the expression 3 + 3 * 2 we want addition to be carried out first, before the multiplication
- To do so we use **parentheses** to enclose the sub-expression that we want to perform first (3 + 3) * 2
- Thus 3 + 3 is carried out first, then 6 * 2 is carried out → 12 (not 9)
- REMEMBER: Parentheses can force any operation to take precedence over all the other operations.
- We can have several sets of parentheses and in this case they are evaluated starting with the inner most set of parentheses: (4 * (3 + 2) – 10) / 10
  1. First, (3 + 2) is evaluated → 5
  2. Then, (4 * 5 – 10) in which case the operator precedence applies: 4*5 → 20 first, then 20 -10 → 10
  3. Last, the division is performed 10 / 10 → 1.0

Arithmetic Expressions and Operator Precedence … your turn

Evaluate the following arithmetic expressions:

```
4 + (5 * (6 / 3))
14.0
4 + ((5 * 6) / 3)
14.0
1 + 2 - (3 * (10/5))
-3.0
1 % 2 - ((3 * 10) / 5)
-5.0
1 % 2 - ((3 * 10) / 5))
SyntaxError: invalid syntax
```

Computational Problem Solving

- Computational problem solving – the use of computational technologies to solve problems;
- One of the most important computational technologies is computer programming and software design.
- **Before** writing a program, it is essential:
  - To understand the **problem** that the program is trying to solve;
  - To understand what are the **building blocks** and how to represent them.
  - To understand if there are more than one **possible solutions** and decide on the most appropriate one.
  - To employ proven **program construction principles**, i.e. structured programming.

- The process of computational problem solving entails the following:

1. Problem analysis
2. Program design
3. Program Implementation
4. Program Testing

Program Design: Abstractions and algorithms

- Any computing problem can be solved by:
  - ✓ understanding the necessary building blocks representing them using only ***relevant*** aspects, and
  - ✓ by understanding what are the necessary steps and executing these steps/actions in a ***specific*** order.

- ➢ Selecting important aspects of the problem and striping away irrelevant details is called **abstraction:**
  - for example, in calculating average GPA: an important aspect is whether the data should be represented using integers or floats, but is the font of output important???
- ➢ Breaking down a problem into a finite number of clearly described, unambiguous and achievable steps that can be systematically followed in a certain sequence to solve a problem in a finite time is called an **algorithm.**

Algorithms

So… **algorithms** entail breaking down a complex problem into smaller steps/actions that must be carried out in very specific sequence/order.

EXAMPLE - going to college

(1) Get out of bed; (2) take off pajamas; (3) take a shower; (4) get dressed; (5) eat breakfast; (6) carpool to college.

- Some steps are compulsory (getting dressed), others may be skipped (e.g. eating breakfast).
- Order is also important, for instance, you cannot (…or better said you shouldn't) get dressed before taking a shower.

- **Both, the number of steps and the order in which they are carried out define an algorithm.**
- Specifying the order in which statements (actions) execute in a program is called program control.

Program Design: Pseudocode

- To develop algorithms we can use pseudocode initially.
- **Pseudocode** is an informal language that helps you develop algorithms without having to worry about the strict details of a language syntax.
- By using pseudocode, the algorithm can be then translated into any programming language.
- So, once we develop an appropriate algorithm, we can then convert to structured portions of Java code or Python code, or C code, and so on...
- Pseudocode is an **artificial** and **informal** language, similar to everyday English, however a lot more structured.
- Pseudocode is user friendly, but it is **NOT** a programming language.
- It helps the programmer "think out" a program before attempting to write it in a computer language such as Python or Java..

Pseudo code example: Adding 2 numbers inputted by a user and display their sum

```
BEGIN
    PROMPT the user to input values for 2 numbers and
    store them into 2 variables;
    STORE the sum of number 1  and number 2 into
    another variable (this is optional);
    DISPLAY the value of the sum.
END
```

EXERCISE: write pseudocode for a program that calculates the GPA for computing year 1, semester 1 (the list of modules and how to calculate GPA is on Course Coordination Page).

Program Design: Activity Diagrams

- We can further visualize the workflow of a program by using an **activity diagram**, such as an **UML activity diagram**;

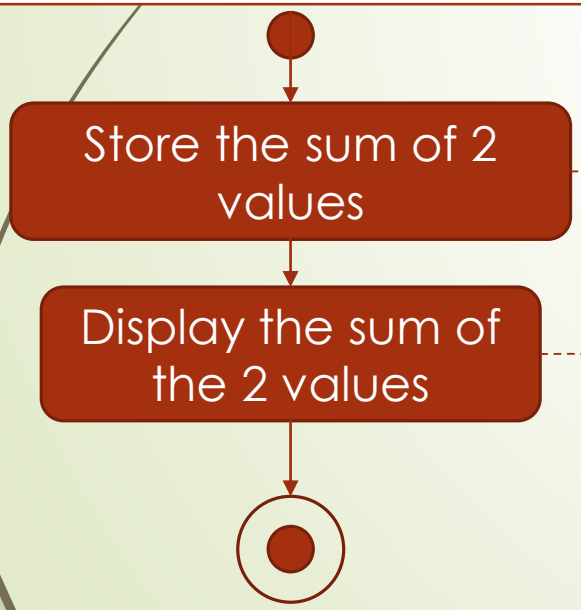
- An activity diagram models the workflow (also called the activity) of a portion of a software system or the entire system.

- Activity diagrams help you develop and represent algorithms/portion of algorithms, because they show how control structures operate.

- It is composed of **symbols such as:**

✓ Rectangles (with their left and right sides replaced with outward arcs);
✓ Diamonds;
✓ Circles;
✓ Arrows.

- Action states represent actions to perform – rounded rectangles.
- Transitions show the order in which the actions represented by the action states occur - arrows.
- Decisions – diamonds.
- The initial state/the beginning of the workflow, before any action has been performed – solid circle at the top of the diagram.
- The final state/the end of the workflow, after the actions have been performed – solid circle surrounded by a hollow circle.
- NOTE: an activity diagram can show only portion of the entire workflow or the entire workflow.



Corresponding python statement:
sum = number1 + number2

Corresponding python statement:
print("the sum of the 2 numbers is", sum)

Exercise:
Draw an activity diagram for the previous GPA exercise

Computational Problem Solving

- Let's apply the problem solving approach to a real problem: write a program that converts temperatures in Fahrenheit degrees to temperatures in Celsius degrees:

1. **The problem understanding** – we need to determine what steps and formulas are required in converting a given Fahrenheit temperature to the corresponding Celsius temperature.

2. **Program design**
    1. **Program requirements**
        1. Inputs: The temperature in Fahrenheit degrees
        2. Outputs: The temperature in Celsius degrees with 2 decimal places accuracy

    2. **Data Representation:** numerical (floating point numbers)

Computational Problem Solving

(program design continued…)

3. **Limitations** (problems that the program cannot address (yet…)):
   1. The program will not address issues with inputs that are not numerical.
   2. The program must be re-run with every input.

4. **The algorithm:**
   1. Prompt user to enter temperature in Fahrenheit degrees and store that input into a variable
   2. Convert the temperature stored in the previous step to Celsius degrees using the formula: Celsius temperature = (Fahrenheit temperature - 32) * 5 / 9
   3. Output the temperature with 2 decimal places.

Computational Problem Solving

3. **Program Implementation** – writing code for the steps outlined by the algorithm:

   1. ## need to ensure that we carry out type conversion because inputs are always strings, but we need float: **fahrenheit_temp = float(input("Enter temperature in Fahrenheit degrees "))**

   2. **celsius_temp = (fahrenheit_temp – 32) * 5 / 9**

   3. **print(fahrenheit_temp, "Fahrenheit degrees =", format(celsius_temp, '.2f'), "Celsius degrees")**

4. **Program Testing/Debugging** – run the program with various values, and address any issues that are not listed as limitations; fix any errors or bugs.

Computational Problem Solving

- **Program Debugging:** the process of finding and correcting errors/bugs in a computer program.

- There are **2 types of errors**:

1. Syntax errors - caused by invalid syntax and they are caught by the interpreter. Examples include:
   - forgetting the closing parenthesis: print('hello'
   - Invalid naming: my+var
   - Etc.

2. Semantic (logic) errors/bugs are caused by errors in the program logic and they are typically discovered at run time. They cannot be automatically detected, because the interpreter does not understand what the script is meant to do: it simply follows instructions. Examples of logical errors include:
   - Calculating average: (n+m+k)/2
   - Calculating average again: num1 + num2/2

Relational Operators, order, and character representation

Relational Operators perform comparisons that evaluate to Boolean values (bool in Python), that is, either True or False.

- These are binary operators that can be applied to any values that have a natural ordering:
  - ✓ Numerical values are typically ordered from smallest to largest
  - ✓ Strings are typically ordered in lexicographical/alphabetical order which is based on the numerical representation of characters

Character Representation:

- To be understood by computers, characters need to be encoded: they need to be numerically represented.
- Unicode encoding typically is considered universal and is a collection of schemes based on 8 and 32 bits for each character.
- Unicode is capable of representing over 4 billion different characters, enough to represent the characters of all languages, past and present
- The default encoding in Python is UTF-8 (which is part of the Unicode standard):
  - is an 8-bit encoding
  - is compatible with the ASCII encoding scheme

Character Encoding

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3				!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~			

- We are interested only in common characters (those from the ASCII table: from 32 to 126).
- Each character has a unique ordinal value which is obtained by combining the row number and column number from the ASCII table:
  - The value of 'a' is 97 (row 9, column 7);
  - The value of 'R' is 82 (row 8, column 2);

NOTE: ASCII set contains the English characters which is a subset of the UNICODE set;

- In Python we can use the built-in functions to convert a character to its underlying numerical value and vice-versa:

- ord function to find out what the value of a characters is: ord('A'); ord('a')
- chr function to convert a positive int to a string character: chr(52); chr(79)

Back to Relational Operators

Relational Operator	Description	Example	Result
== (Equal to)	Tests whether two values are equal	10 == 10 'String' == 'string'	True False
!= (Not Equal to)	Tests whether two values are not equal	10 != 10 'String' != 'string'	False True
< (less than)	Tests whether a value is less than another value, or whether it comes before another value	10 < 20 'String'<'string'	True True
> (greater than)	Tests whether a value is greater than another value, or whether it comes after another value	10 > 50 'strong' > 'string'	False True
<= (less than or equal to)	Tests whether a value is less than or equal another value, or whether it comes before or is equal another value	10 <= 7 'String' <= 'string'	False True
>= (greater than or equal to)	Tests whether a value is greater than or equal to another value, or whether it comes after another value or is equal another value	10 >= 10 'string' > 'strong'	True False

Some Exercises…

- Use the ASCII table to do the following:
  - Find the value of the character '&'
  - Find the value of the character "'"
  - Find the character with the value 77
  - Find the character with the value 122
  - For each of the above also use the corresponding built-in function to test that your results are correct.
- What are the values of the following comparisons?
  - 10 == 10
  - 10 == 10.0
  - 10 != 11
  - 'Strong' > 'strong'
  - 'hello world' >= 'hi'
  - 34 <= 37
  - 'hello' <= 'hi'
  - 'hello' <= 'hello'
  - 'hello' == "hello"
  - '7' == 7

Another useful built-in function…

- The relational operators check only for lexicographical order when applied directly to strings, that is, they check if a string comes before or after another string.
- To check whether a string is longer than another, that is, whether it contains more characters, we need to compare their lengths, not their lexicographical representation;
- To obtain the length of string we can use the function ***len*** which returns the number of characters the string enclosed within its parenthesis has:

```
len('hello') ## returns 5
len('hello world') ## returns 11
len('a') ## returns??
```

- So… len('hello') < len('hello world') returns True, while len('a') > len('hello') returns False;

EXERCISES: What will the following expressions return?

len(''); len(" "); len('3'); len('abc') == len('xyz'); len('man')>len('men'); len('45') >= len('100'); len('Anna') != len('Dana'); len(2) <= len(2.1)

More Operators … Membership Operators

- Python also provides **2 operators to test membership**, that is to test whether a value is part of a list/set/sequence/etc. of other values.
  - **in** - returns **True** if the specified value is the given list/set/sequence/etc.
  - **not in** - returns **False** if the specified value is not the given sequence/etc.
- These operators are also binary operators
- EXAMPLES:

```
'Aurelia' in 'Aurelia Power' ## returns True
3 in (4, 7, 3, 21, 77)## returns True
 'FOP1' not in 'Aurelia Power' ## returns True
'Aura' in 'Aurelia Power' ## returns False
11 in (4, 7, 3, 21, 77)## returns False
3 not in (4, 7, 3, 21, 77)## returns False
'Power' not in 'Aurelia Power' ## returns False
```

- For now, we will only test membership for string sequences, that is to see if a string occurs in another string.
- EXERCISE: specify what the following expressions return:

  '2' in '4'; 'hello' in 'Hello world'; 'Maths' not in 'Programming'
  '2' in '22'; 'Character' not in 'Characters'; '3' not in '3+4'

More Operators … Boolean/Logical Operators

- There are three Boolean operators in Python;
- These operators apply only to **Boolean values**, in other words, the operands must of bool type;
- They also return either True or False:

  and – is a binary operator that returns True only when both operands are true, otherwise False

  or – is a binary operator that returns True when either operand is True, and False when both operands are False

  not – is a unary operator that simply reverses the truth value of its operand: not False equals True, and not True equals False.

X	Y	X and Y	X or Y	not X
False	False	False	False	True
True	False	False	True	False
False	True	False	True	
True	True	True	True	

Logical/Boolean Expressions

- Boolean operators are used to build complex Boolean expressions:
  1 < 2 and 2 < 1 ## False
  3 + 1 == 2 or 3 + 1 == 4 ## True
  'str' in 'string' and 7>=5 ## True
  not 'hello'=='hallo' and 'hallo'< 'hello' ##True

- NOTE: Python uses short-circuit/lazy evaluation for Boolean expressions: the second operand is evaluated only if the truth value cannot be determined from the first operand alone

- Examples:
  - 'str' == 'string' and 'hello' > 'hi' – only 'str' == 'string' is evaluated because it is False, and if any operand of the logical ***and*** is False, the entire expression is False
  - 2 > 1 or 'hi' == 'hello' – only 2 > 1 is evaluated because it is True, and if any operand of the logical ***or*** is True, the entire expression is True

Equivalent Expressions

- In maths, there are many arithmetically equivalent expressions of different form; for instance:
  x*(y+z) is equivalent to x*y + x*z for any values of x, y and z

- Similarly, there are logically equivalent Boolean expressions of different form; for example, for any values of x and y:

  x < y is equivalent to **not(x >= y)**
  x <= y is equivalent to **not(x > y)**
  x == y is equivalent to **not(x != y)**
  x != y is equivalent to **not(x == y)**
  not (x and y) is equivalent to **not(x) or not(y)**
  not (x or y) is equivalent to **not(x) and not(y)**

- The last 2 are known as De Morgan's Laws

Operator Precedence Revisited…

Precedence level	Operator(s)	Associativity
First	** (exponentiation)	Right – to – left
Second	- (negation)	Left – to – right
Third	* (multiplication), / (division), // (floor division), % (modulus)	Left – to – right
Fourth	+ (addition), - (subtraction)	Left – to – right
Fifth	Relational Operators: <, >, <=, >=, !=, ==	Left – to – right
Sixth	Logical *not*	Left – to – right
Seventh and eight	Logical *and* and logical *or*	Left – to – right

Exercises

- Evaluate the following Boolean expressions using the operator precedence rules of Python.
  10 >= 8 and 5 != 3
  10 >= 8 and 5 == 3 or 14 < 5

- What is the value of variable **num** after the following is executed?
  num = 10
  num = num + 5
  num == 20
  num = num + 1

- Which one of the following Boolean expressions is not logically equivalent to the other two?
  not(num < 0 or num >10)
  num > 0 and num < 10
  num >= 0 and num <= 10