Fundamentals of Programming 1

Lecture 2

Aurelia Power, ITB, 2018

The Basics of the Python Language

- Each programming language has a syntax which defines a set of rules for well formed statements/instructions:
  - ✓ Python has a simple syntax → highly readable language
- Python is a dynamically typed language:
  - ✓ The **value** of a variable carry the datatype, not the **variable** itself.
  - ✓ No need to declare data type for a variable.
- The core of Python programming are objects: everything in Python is an object.
- The primitive constructs of Python include literals, variables, and operators.
  - ➢ These can be combined to form expressions
  - ➢ Each expression is also denoted by an object of some type.

The Basics of the Python Language

- Every object has a type that are divided into:

1. **Scalar (indivisible) types** = cannot be broken further into subcomponents;

- For example: the integer 12 cannot be further broken down into subtypes
- Scalar types include: integers, floating point numbers, Booleans etc.

2. **Non-scalar types** have internal structures that can be further broken down;

- For example, a String is sequence of characters, so it can be further broken down into subcomponent types (characters).

- To find out the type of an object we can use the built-in function **type.**

```
type(12) ## <class 'int'>
type('fop1') ## <class 'str'>
```

Note: **in-line comments** in Python are denoted by the # symbol; comments are ignored by the interpreter and allows us to document our programs.

Exercise: in the shell, use the type function with several values

Important Types in Python

Data type	Examples	Description	Scalar or not	Python Representation	Using the type function	Output
Integers	1 or -7	Whole numbers	Yes	int	type(1)	<class 'int'>
Floating point numbers	3.4 or -7.7	Real numbers	Yes	float	type(3.4)	<class 'float'>
Boolean	Only 2 possible values: **True** and **False**	Booleans	Yes	bool	type(True)	<class 'bool'>
None	Only one value: **None**	Used in Python to represent nothing/empty state	Yes	None	type(None)	<class 'NoneType'>
String	'Hello World' "Python"	A sequence of characters	No	str	type('hello')	<class 'str'>

Literals in Python

- A literal = a sequence of one or more characters that stands for itself.
- Literals are values.
- Numeric Literals: literals containing only the digits 0-9, and/or + or – signs, and/or the decimal point; commas are never used.

Examples:

- Integer literals: 1, -10, +12
- Floats literals: 1.0, -10.3, +12.7

NOTE: We can use the scientific notation to represent large floats

9.453 X 10^{8} (945300000.0)can be represented in python as 9.453e+8

9.453 X 10^{-8} (0.00000009453) can be represented in python as 9.453e-8

EXERCISE: in the shell type several numerical values, including using scientific notation.

Literals in Python

EXERCISE: let's multiply and divide very large floats using the * symbol for multiplication and / for division

>>> 1.2e200 * 2.4e100 ## (outputs 2.88e+300)

>>> 1.2e200 / 2.4e100 ## (outputs 5e+99)

>>> 1.2e200 * 2.4e200

>>> 1.2e2200 / 2.4e20

The last 2 examples give inf because they exceed the range for floats.

- there are no limits to the size of an integer in Python, but there are for floats both in terms of range and precision:
- Float range: $10^{-308}\ to\ 10^{308}$
- if a value is outside this range, arithmetic underflow or overflow can occur, that is, the calculated result is too small or too large to be represented in python.

Literals in Python

EXERCISE: mathematically, the result of dividing 1 by 3 is 0.3333333333… with 3 repeating infinitely after the decimal point; let's do this in the python shell:

>>> 1 / 3

0.3333333333333333

… so only 16 digits have been outputted after the decimal point because there can only be a finite number of digits that can be displayed.

- Float precision: the number of digits that are displayed after the decimal point is 16.
- Since any floating-point representation contains only a finite number of digits, what is stored for many floating-point values is only an approximation of the true value.
- Conversely, if we multiply 3 * (1/3), mathematically we should get 0.99999999999999, but in python we get..

>>> 3 * (1 / 3)

1.0 – because python is rounding up the result

EXERCISE: in the shell type in the following:

1/10, 1/10 + 1/10 + 1/10, 10 * (1/10), 6 * (1/10), 6 * 1/10

The built-in function format for formatting floats

- The results of float calculations can contain any number of decimal places

```
>>> 12/5
2.4
>>> 5/7
0.7142857142857143
```

- We can use the built-in function **format** to restrict how many decimal places we want to be displayed in the output:

```
>>> format(12/5, '.2f')
'2.40'
>>> format(5/7, '.2f')
'0.71'
```

Format specifier for decimal places
- in this case it specifies only 2 decimal places

The built-in function format for formatting floats

- NOTE: the format function always returns a string, not a numerical value.
- For instance it returns '2.40', not 2.40 (note the quotation mark that enclose the output)
- You can also specify separators, for example the comma can be used to separate thousands.

```
>>> format(13402.2511, ',.2f')
'13,402.25'
```

EXERCISE: in the shell type in the following:

format(11/12, '.2f')

print("the cost of repeating the FOP1 module is: ", format(23555.55555, ',.2f'))

NOTE: python rounds up before outputting

NOTE: each parenthesis, square bracket, curly brace and quotation mark must have a matching closing counterpart, otherwise you will typically get a SyntaxError: invalid syntax

String Literals in Python

- String literals are represented as character sequences (including letters, digits, special characters and spaces enclosed within **quotation marks** (either double or single):

  'C' – a string containing a single character

  'this is the FOP 1 module' – a string containing letters and a digit

  "aurelia.power@itb.ie" – a string containing letters and special character

  "" – the empty string

  " " – a string containing the blank/space character

- All strings must be enclosed within matching quotation marks:

  >>>"Hello ITB'

  SyntaxError: EOL while scanning string literal

  >>>'Hello ITB"

  SyntaxError: EOL while scanning string literal

String Literals in Python

- Strings can also contain quotation marks:

```
>>> “A student’s gpa is calculated using…”
'A student’s gpa is calculated using…'
>>> ‘A student’s gpa is calculated using…’
SyntaxError: invalid syntax
>>> “The student said “I love Python””
SyntaxError: invalid syntax
>>>  ‘The student said “I love Python”’
'The student said "I love Python“'
```

- We can also use escaping if we want the same type of quotation marks:

```
>>>'The student said \'I love Python\"
"The student said 'I love Python’"
>>> "The student said \"I love Python\""
'The student said "I love Python“'
```

Escaping in Python

- **The escape character** is the backwards slash \
- It used to create **escape sequences** (which are also represented as strings and must be inside quotation marks, double or single) such as:

- Escaping double quotation: \" – "the man said \"hello\""
- Escaping single quotation \' – 'the man said \'hello\''
- Escaping the new line \n – 'the man said\nhello'
- Escaping the horizontal tab \t – 'the man said\thello'
- Escaping the backslash \\ – 'the man said \\hello\\'

NOTE: some sequences are outputted literally when typed directly in the shell; they must be used in a function such as *print* to obtain the underlying representation.

Note: if you want to type more than one command on the same line you must separate them using the semicolon: print(3); print('fop1')

Back to String Literals in Python…

- Strings must be contained on only one line if using single or double quotes
- But we can contain strings on multiple lines using triple quotes:

```
"""Welcome to ITB!!
It is an exciting time for the first year computing students, especially in FOP1 ☺!!"""
```

NOTE: multiple line literal stings are outputted in the shell using the new line character whenever a new line is started.

EXERCISE: in the shell type in the following commands and explain the output

```
print('hello'); print('hello"); print('She said\'hello\'.')
"print the following lines: \nline1\nline2\nline3"
print("the \\ is used to create escape sequences")
'''hi
hi
'''hi
```

Basic String Formatting in Python

- We can use the built-in function to control how strings are displayed.
- The general form of the *format* function is:

  format(value, format_specifier)

- Left (also the default), centered and right justified format specifiers: <, ^, and >
  - Left justified in a field of 30 characters: format('FOP1', '<30')
  - Right justified in a field of 30 characters: format('FOP1', '>30')
  - Centered in a field of 30 characters (15 on each side): format('FOP1', '^30')
- Filling characters specifier - just the number of characters: format('*', '70')
- Combining formatting: print('Python this semester', format('-', '<30'), 'Java next semester'))

NOTE: the print function can take any number of arguments all separated by coma

- **EXERCISE:** type in the shell and explain.

```
print(format('First Name: Aurelia', '^70'))
print(format('2','2<10'), 'Surname: Power', format('2','2>10'))
```

Implicit and Explicit Line Joining in Python

- The Python-recommended maximum length for a line is of 79 characters.
- Sometimes we need to fit more characters and to span our code over several lines → **2 ways** in Python to do deal with such situations

1. implicit line joining - matching parentheses, brackets, braces and triple quotes allow spanning over more than one physical line:

```
>>> print('hello',
          'hello')
>>> print(3+4,
          'equals',
           7)
```

2. explicit line joining - program lines may be explicitly joined by use of the backslash (\) character:

```
>>> 3 + 4 + \
    3 + 4 + \
    3 + 4
```

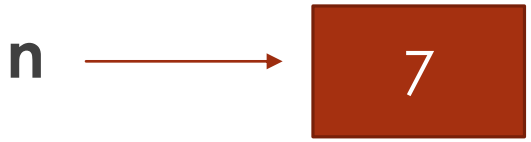
Variables

- So far we have used literal values in our programs.
- But many times we need to store such values so we can use them later, and to do so we use variables:

Variables are used to store values

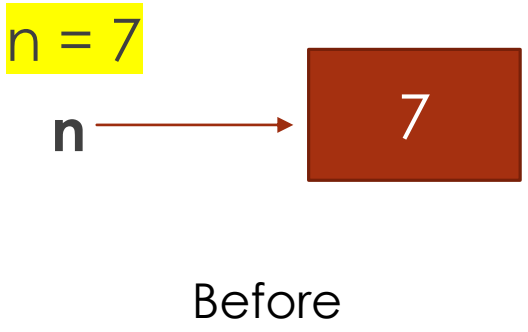
A variable is a named location in the computer's memory that points to a given value; so, each variable has a name (aka *identifier*) that is associated with a value.

- for example, the command n = 7 will create a variable with the name n that will point to the value of 7

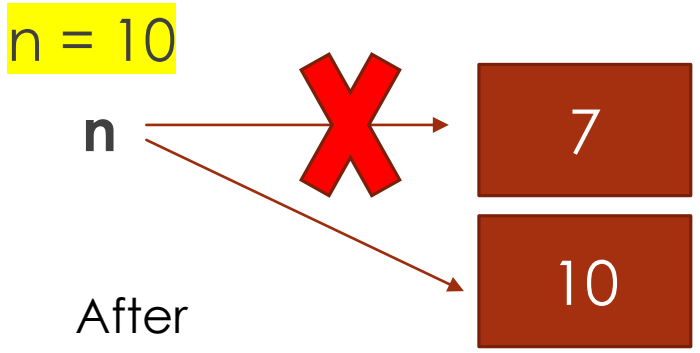


Variables

- **NOTE**: when used the first time, a variable must be associated with a value, otherwise you will get a NameError: name 'n' is not defined
- So we must use the **=** operator (the equal operator) in programming to associate values with variables (not to test for equality).
- The process of associating names with values is called **assignment**.
- The **named locations** that are the result of assignment are called **variables** because their values are allowed to *vary* over the life of the program → we can assign different values to a variable during the lifetime of a program



Before



After

Variables

- **EXERCISE**: type the following commands and explain.

  n = 7.3; lang = 'python'

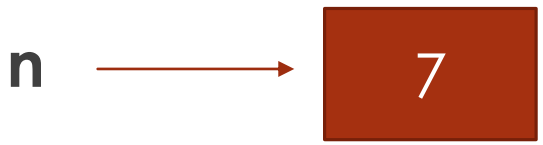
  n; lang

  n = 10.2; lang = 'java'

  n; lang

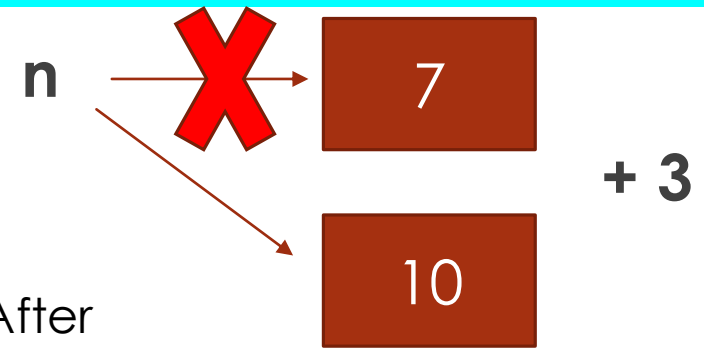
- We can use the old value of a variable for the re-assignment

n = 7



Before

n = n + 3 (new value of n = old value of n which is 7 + 3)



After

Variables

- **EXERCISE**: type the following commands and explain.

```
n = -2; lang = 'python'
n; lang
n = n + 3; lang = lang + 'java'
n; lang
```

- **NOTE:** when we use the **+** with numerical values the interpreter carries out mathematical addition, but when used with strings, it carries out string concatenation: the process of putting together 2 strings;
- This phenomenon is called operator overloading: it carries out different operations depending on the type of the operands.

```
3 + 4 ## 7
'FOP ' + '1' ## 'FOP 1'
'FOP ' + 2
TypeError: must be str, not int
```

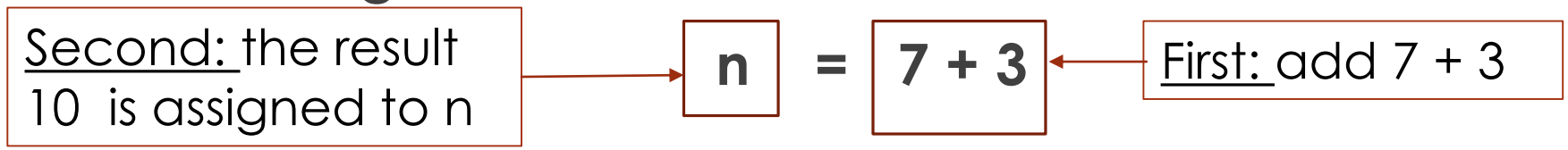
We cannot concatenate a str and an int, nor can we add an int with a str!!!

- Another example of operator overloading is the multiplication operator: *

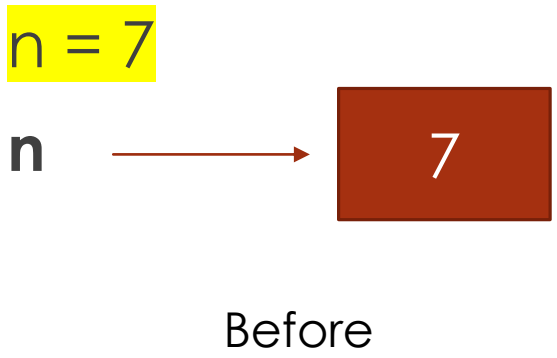
```
3 * 4 ## 12
3 * '4'

## '444' because multiplication can be thought in terms of addition/concatenation: '4' + '4' + '4'
```

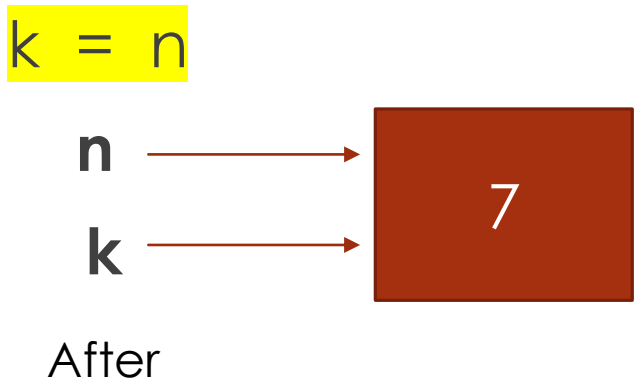
Variables … more on assignment

- The right hand side of the assignment is evaluated first, then the result or the value is assigned to the variable on the left hand side:



- Variables may also be assigned to the value of another variable:

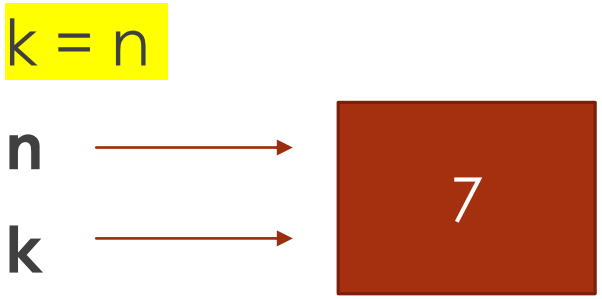




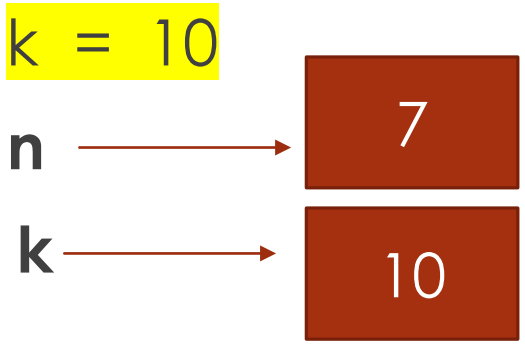
- Variables n and k are associated with the same literal value of 7 in memory

Variables … more on assignment

- So variables can share the same value → if we change the value for one variable, does it mean that it is also changed for the other variable??
- NO because integer values are immutable → literal integer values cannot be changed
- In fact, all types that we have discussed so far: int, float, bool, and str are <u>immutable</u> (cannot be changed) in python



Before



After

- <u>NOTE</u>: if no variable refers any longer to the memory location of the original value, it is de-allocated/made available for re-use

Variables … more on assignment

- **In Python the same variable can be associated with values of different types during the lifetime of a program:**

  my_var = -3 ##(integer)

  my_var = 3.4 ##(float)

  my_var = True ##(Boolean)

  my_var = 'FOP1' ##(String)

- **EXERCISE:** after executing the code below

n = 10; k = 7; j = -12; l = 'line'

n = k + j; j = k; l = l + l; k = l * 2

What are the values of n, k, j, and l?

Explain.

Naming your variables

- Pick a name that explains its purpose: num_of_students, kmPerHour, etc.
- **Rules** for naming variables:

1. Variable names can be of any length
2. Variable names must start with a letter or the underscore, and the remaining characters must be letters, digits, or underscores.
3. You cannot use other symbols such as ?, ! , +, - , * or % because they are already used as operators.
4. Spaces are not permitted inside names either.
5. You can use uppercase letters or underscores to denote word boundaries, as in cansPerPack or cans_per_pack (typically in python underscores are used, whereas in java typically the camel case convention is used… but they are just conventions).
6. Variable names are case sensitive: canVolume and canvolume are different names.
7. You cannot use reserved words such as and or as

Reserved Words in Python that cannot be used as identifiers (you don't need to learn them by heart…)

- and
- as
- assert
- async
- await
- break
- class
- continue
- def
- del
- elif
- else
- except
- exec
- False
- finally
- for
- from
- global
- if
- import
- in
- is
- lambda
- None
- nonlocal
- not
- or
- pass
- print
- raise
- return
- True
- try
- while
- with
- yield

Operators in Python

- **Operator** = a symbol that represents an operation that may be performed on one or more operands

1. **Unary Operators** – take only 1 operand
    1. The logical *not* and the bitwise *not* – not covered today
    2. The negative sign
2. **Binary Operators** – take 2 operands
    1. All mathematical operators
    2. Logical *or* and logical *and* – not covered today
    3. Comparison operators – not covered today
    4. Assignment operators – not all covered today
    5. Identity Operators – not covered today
    6. All membership operators – not covered today
    7. Most bitwise operators – not covered today
3. **Ternary Operators** – take 3 operands
    1. Conditional expressions – not covered today

Arithmetic Operators

Operator	Name	Example	Notes
- (unary)	Negative sign	- 12	It negates the sign of the given value
+ (binary)	Addition	7 + 3	Addition of ints gives an int result; addition of floats gives a float result; addition of a float and an int (and vice versa) gives a float result.
- (binary)	Subtraction	7 - 2	Same as above.
* (binary)	Multiplication	7 * 2	Same as above
/ (binary)	Division	7 / 2	Always gives a float result.
% (binary)	Modulus(Remainder)	7 % 2	Same as addition, subtraction and multiplication.
** (binary)	Exponentiation (Power)	7 ** 2	Same as addition, subtraction, multiplication and modulus.
// (binary)	Floor Division (truncation – discards all digits after the decimal point)	7 / 2	Same as addition, subtraction, multiplication, modulus and exponentiation.

What are the data type of the results of the following operators???

>>> 1 + 3.4 float

>>> 1 - 5 int

>>> 4 * 2 int

>>> 2.1 * 2 float

>>> 3/1 float

>>> 7 ** 2 int

>>> 2 ** 3.1 float

>>> 7 % 3 int

>>> 7.3 % 2 float

>>> 2//3 int

>>> 2.1 // 3.1 float

>>> 2.1 // 3 float

- How many binary operators are in the following expressions?

  -10 + (-23) one

  -12 * 1 + 10 – 1 three

- Give the exact results of each of the following division operations:

  5 / 4 1.25

  5 // 4 1

  5.0 // 4 1.0

Basic Keyboard Input

- In many programs the values of variables are decided by the user
- To take input from the user we can use the built-in function ***input***

EXAMPLE:

```
>>> name = input( "What is your name? “ )
What is your name? Aurelia Power
>>> name
‘Aurelia Power’
```

NOTE: the input function always returns a string value; so if we need numerical values we need to explicitly convert them using the built-in functions: *int* and *float;*

- The process of explicitly converting the type of a value is called **type conversion.**

EXAMPLE:

```
>>> age = input( "What is your age? “ )
What is your age? 21
>>> type(age)
<class 'str’>
```

Basic Keyboard Input

EXAMPLE (continued):

```
>>> age = int(age)
>>> type(age)
<class 'int'>
>>> gpa = input( "What is your gpa? " )
What is your gpa? 4.0
>>> type(gpa)
<class 'str'>
>>> gpa = float(gpa)
>>> type(gpa)
<class 'float'>
```

EXERCISE: Take 3 inputs from the user asking them for their age, then output the average of their age.

1. Ask for input and store in a variable (do this 3 times)
2. Convert each input
3. Add them up and divide them by 3
4. Then use the print function