2: Variables and expressions

Data manipulation

- Programs are written to manipulate data.
- To accomplish this, the data needs to be stored into computers memory somehow.
- The data in Python is stored as **objects.**

Python and objects

- Objects are hence "chunks" of data, such as
  - Integers, such as 3, -1 or 56000
  - Floating point numbers, such as 4.2, -0.05 or 100.0
  - Strings, such as "a", "Hello" or "This is a sentence"
- There are bunch of other types as well, we return to this later...

Objects (2)

- Objects have **type**, **identity** and **value**

- **identity**
  - Property that distinguishes a particular object from all other objects, even if they had the same type and value.

- **type**
  - Tells the type of the object’s value. Remains constant during its lifetime. You can also define own types, so the number of possible types is almost infinite.

- **value**
  - The data stored in the object. Depending on the object type, the value **may** change during the lifetime of the object.

References

- a **reference** is, for our purposes, a piece of information that identifies an object

- Think of reference numbers on sample tubes, in libraries, etc. . .

Handling data in programs

- To manipulate data in programs, the data needs to be accessed somehow.
- References are not very convenient, as they are typically memory addresses.
- Hence, programming languages utilize **variables.**

Variables

- A **variable** in Python is a *named reference to an object*.
- With variable, it is possible to access the data referenced by it.

Creating variables

- The most usual way of **defining** a variable in Python is to assign an object (i.e. a **value**) to it.

- *(Often value is a reference to object; this is, however, discussed later in this course)*

Assigning variables

- Variable **type** is determined dynamically, when assignment is done:

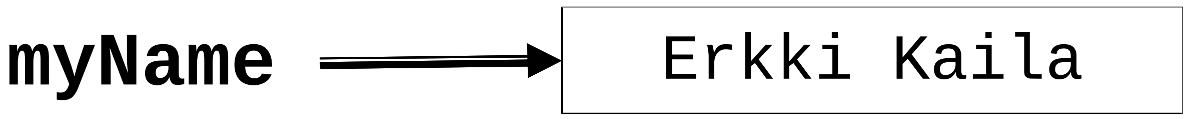
```
    myName = "Erkki Kaila"
```

Assigning variables (2)

- The statement creates an object of a type *string* with the value "Erkki Kaila"
- Defines a variable called myName, in which a *reference* to object created is saved.
- now, whenever you refer to the variable myName, the reference to object with value "Erkki Kaila" is returned

Assigning variables

- This can be illustrated with an image where **variable** references an **object**.



Referencing object (values):

- `myName = "Erkki Kaila"`
- `print myName`
- Fetches the value of variable `myName`..
- ..and outputs **`Erkki Kaila`**

Types of objects

- Python supports several types of objects (and it’s even possible to define your own types).
- We start with some basic types, and later expand the selection when needed.

Types of objects

Type	Examples
Integer number	1 35003 -123456
Floating point number	0.22 -12.500003
String	"Hello" "This is a longer string"
Boolean	True False

Assigning variables with different types

- Variable with integer value:

```
    myNumber = 23
```

- Variable with floating point value:

```
    b = 1.03
```

- Variable with **boolean** value:

```
    thisIsTrue = True
    otherVariable = False
```

Variable assignment syntax:

- Hence, the variables are always assigned using the following syntax:

```
variableName = <expression>
```

Variable names

- a variable name:
  - Always starts with a letter (A...Z, a...z)

  - Can contain only letters, numbers and underscore _

  - is **case sensitive**, meaning that myvariable is **not equivalent to** MyVariable

  - is not a reserved word in Python (see section 2.3.1 of Python language reference)

Examples

- Valid names:
  - `firstName`
  - `first_name`
  - `DNASequence`
  - `tmp1`
  - `Tmp1`
- Invalid names:
  - `1variable`
  - `my name`
  - `first-name`
  - `#ofStrings`

About naming

- There are a couple of widely used **naming conventions** for programming

- I tend to use something called **camel case:**
  - Variable name starts with a lower case letter

  - Consecutive words are written without underscore and with a capital first letter

Camel case naming

- For example
  - `name`
  - `firstName`
  - `nameAndAddress`
  - `veryLongVariableNameWithSeveralWords`

Variable names (2)

- Variables should be named to imply their *intent* and *meaning*
- Consider the following:

```
a = "M. Murdock"
b = 35
c = 75.3
```

- ...and the following:

```
name = "M. Murdock"
age = 35
weightInKG = 75.3
```

Changing variable value

- Variable value can be changed:

```
value = 1
print value # Outputs 1
value = 25
print value # Outputs 25
value = value + 3
print value # Outputs 28
```

Changing variable value (2)

- Changing a variable only affects that variable with no connection to others:

```
x = 1
y = 2
x = y # x == 2 and y == 2
y = 5 # x == 2 and y == 5
```

Assigning variable with other variable

- Variable assignment can contain any type of objects, including other variables:

```
myName = "M. Spencer"
otherName = myName
```

Assigning variable with other variable.

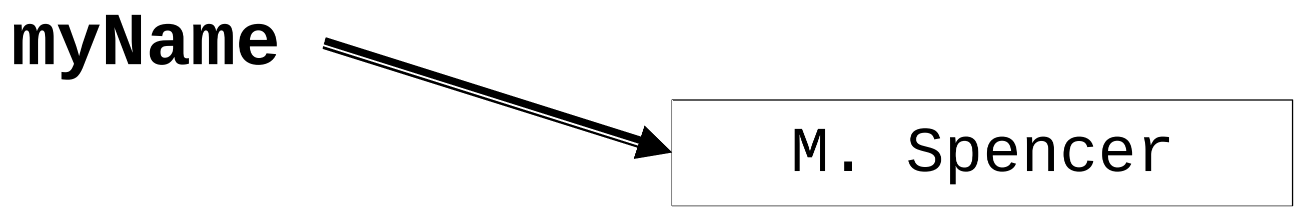
- Note, however:

```
myName = "M. Spencer"
otherName = myName


myName = "S. Stevenson"
```

- ...only changes the reference of the variable myName, leaving otherName intact.

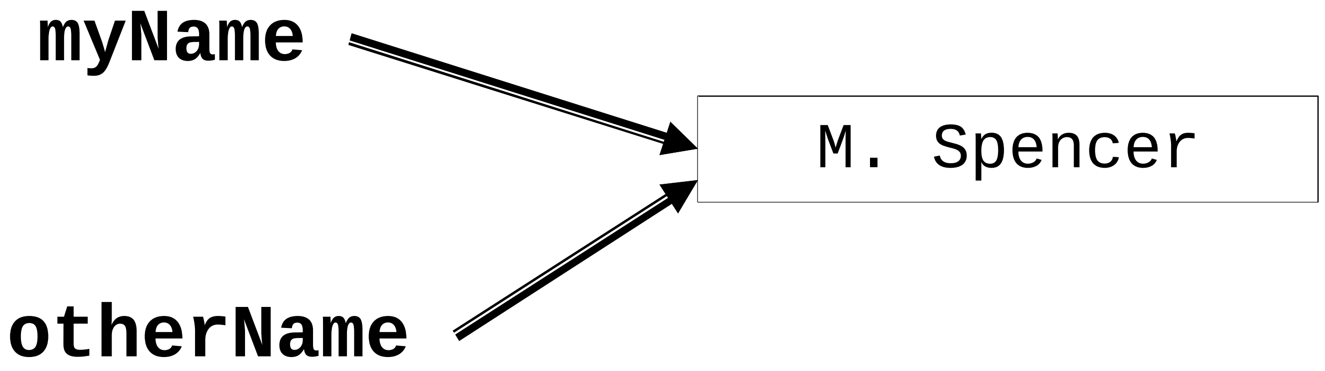
Assigning variable with other variable.

```
myName = "M. Spencer"
otherName = myName

myName = "S. Stevenson"
```



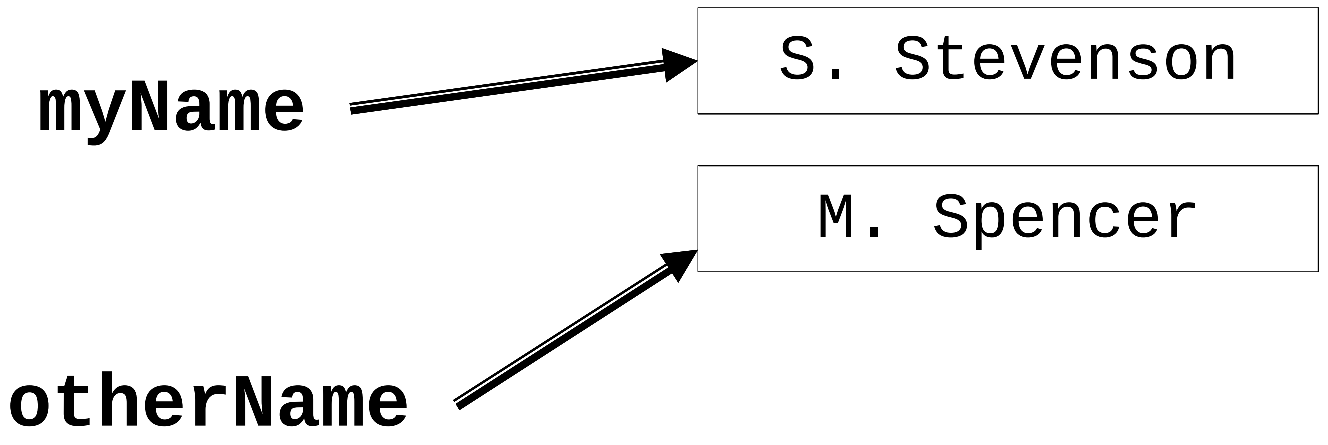
Assigning variable with other variable.

```
myName = "M. Spencer"
otherName = myName

myName = "S. Stevenson"
```



Assigning variable with other variable.

```
myName = "M. Spencer"
otherName = myName

myName = "S. Stevenson"
```



Types of variables (2)

- Note, that though the type of an object remains constant, type of a variable can change:

```
myVariable = "Car"
myVariable = 4
myVariable = 6.0
```

- However, this is *highly unrecommended!*

Assigning several variables at once

- It is possible to assign several variables with one statement:

```
    a, b = 3, 6
```

- ...which is similar to

```
    a = 3
    b = 6
```

Assigning several variables at once

- Usually it is better to assign variables in their own line.

- However, the feature has a special usage. The statement below swaps the values of two variables:

```
    a, b = b, a
```

Variables must be initialized before using them!

- Variable must be initialized (by assigning a value) before using them:

```
print helloWorld
```

→ produces an error, if variable helloWorld is not assigned a value before the statement

Expressions

- An **expression** is a combination of *objects* and *operators*
- What operators do depends on the type of their **operand objects**
- **+** is arithmetic addition if its operands are numbers, but string concatenation if its operands are strings

Expressions (2)

- Expressions are **evaluated (i.e. executed in a fixed order).**

- As a result of the evaluation, expression generates (at least one) new object.

Arithmetic operators

Operator	Explanation	Example
**	Exponentiation	2 ** 3 = 8
*	Multiplication	3 * 2.2 = 6.6 "ab" * 3 = "ababab"
/	Division	5 / 2 = 2 (!!) 5.0 / 2 = 2.5
//	Forced integer division	5.0 // 2 = 2.0
%	Modulo (gives remainder)	5 % 2 = 1
+	Addition	5 + 3 = 8 "aa" + "bb" = "aabb"
-	Subtraction	8.0 – 3 = 5.0

Expressions (cont.)

- Precedence of operators as in mathematics. Use parentheses to indicate precedence:
  - 2**3+1 gives 9 and 2**(3+1) gives 16
  - 2 + 1 * 3 gives 5 and (2 + 1) * 3 gives 9
- More operators are introduced later as needed.

About division

- For historical reasons, the division operator / works as **integer division** when its operands are integers, and as a **real division** when at least one of its operands is a real number:

  5 / 2 results to 2
  5.0 / 2 results to 2.5

- Note, that the // operator forces the integer division:

  5.0 // 2 results to 2.0

About division (cont.)

- Common source of errors
- Works differently in newer (3+) versions of Python
- Magic trick: Use

```
    from __future__ import division
```

- as *the first line of your program* to force correct division.

Statements

- A **statement** is an instruction Python can execute. Think of statement as **ordering Python to do something**.

- We have already seen the **print** statement. It evaluates an expression, and outputs its result:

```
print "Hello world!"
print 2 + 4 * (3 – 1)
```

Concatenating strings

- As seen before, strings can be concatenated with the **+** operator.

```
myName = "John Smith"
print "My name is " + myName

first = "Jane"
last = "Doe"
name = first + " " + last
print name
```

Concatenating strings and numbers

- Strings and other types of objects cannot be concatenated with the + operator
- However, it is possible to output several objects with a single print statement by separating the objects with comma:

```
a = 20
print "The result is", (a * 2), "."
```

Concatenating strings and numbers (2)

- Note, that comma operator doesn't actually concatenate objects; rather, you can list a number of objects for print statement to output:

```
# output all following
print "hello", 23, 1.0, "all"
```

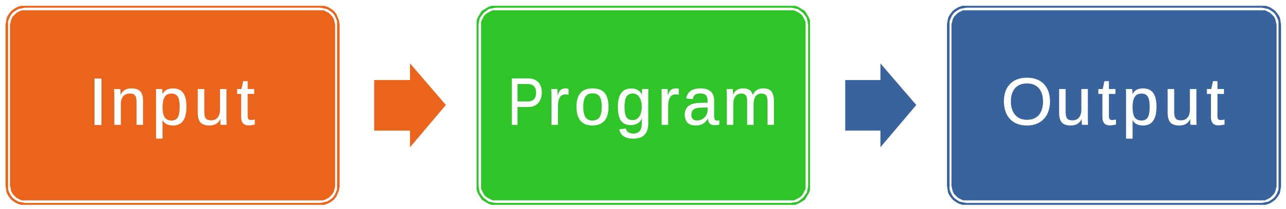
Concatenating strings and numbers (3)

- Hence, you **can not use the comma operator in expressions to concatenate objects:**

```
# THIS DOES NOT WORK AS INTENDED
res = "Result is", 23

# WORKAROUND (discussed later)
res = "Result is" + str(23)
```

User input

- Usually in programs, some kind of user input is required



User input in Python

- Python has two built-in **functions** that can be used to reading values from the user:

- Function **input** is used to read numbers and

- function **raw_input** to read strings.

Functions (briefly)

- Function is something that can be called in a program and which returns a value
- Function can be called as a single statement or as a part of an expression
- If the function result is needed later, the return value must be stored into a variable

Using input function

- **input** function returns the result of expression (such as number) typed in by user:

```
myVariable = input ("How old are you? ")
print "You are", myVariable,"years old"
```

- Note the parentheses used when calling a function:

  input (questionString)

Using input function (cont.)

- Note, that the input function **can not be used to input strings**.

- However, users do make (intentional) errors. We will later return to the methods of *validating the user input*.

Example

- Program that queries the weight in kilograms and displays it in pounds:

```
weightInKg = input("Weight in kilograms :")
weightInLbs = weightInKg * 2.2
print "That is ", weightInLbs, " pounds"
```

Using the raw_input function

- **raw_input** function returns a **string** typed in by user:

```
name = raw_input("Type in your name :")
print "Nice to meet you, " + name + "!"
```

Example 2

- Program that queries first and last name separately and concatenates them in one string, which is then output.

```
first = raw_input("First name :")
last = raw_input("Last name :")
name = first + " " + last
print "Name is " + name
```

Comments

- Most programs are quite complex and it’s often tough to see what they do
- It is good style to *comment* the code so that it’s easier to comprehend
- Comments in Python are denoted by the # character
- The interpreter ignores anything starting with # until the end of the line
- I will insist that you *comment your code properly!*

Comments (cont.)

- Example:

```
first = "David" # assign first name
second = "Jones" #assign last name
name = first + " " + second # add separator
weightInKg = 75

#Convert the weight to pounds
weightInLbs = 2.2 * weightInKg
```