

INF1-IP

Introduction to Programming

Lecture 2 – Strings and numbers



> Types

Like most languages, in Python *values* have *types*.

For example:

- Integers (`int`): whole numbers like 3 and -157
- Floating numbers (`float`): decimal numbers like 7.5 or -4.0
- Strings (`str`): text like "Henry" or "3"

The function `type()` tells you what type a value has

> Values and Variables

Values can be assigned to variables

```
x = 2 # x refers to an integer
```

In Python, variables do not have fixed types

```
x = 2 # x refers to an integer  
x = "hello" # x now refers to a string
```

Python is a *dynamically typed* language – which means that variables can change their type.

Some languages (e.g., Java) are *statically typed* – which means that variables can only have one type.

> Types and classes

The *type* an *object* has is defined by the *class* it is an instance of.

Basic types are built-in – e.g., integer, string, etc.

This happens for things that are frequently used and clearly defined

But often we want to create our own type of thing – e.g, car, student, ...

These are created within **classes**.

e.g.,

- an object of class `int` has the type `int` – *this is a built-in class*
- an object of class `student` has the type `student` – *this is a created class*

> Operators

We have already looked some operators.

Common ones include    

Operators work with objects of certain types. For example, these common ones work with *integers* or *floats*

Some operators work with more than one type, and then they behave slightly differently.

If they get input of the wrong type, a `TypeError` occurs

> Input types of common operators

	Operator	int & int	int & str	str & str	
print(")	+	Addition	TypeError	Concatenation	)
print(")	-	Subtraction	TypeError	TypeError	)
print(")	*	Multiplication	Repeat String	TypeError	)
print(")	/	Division	TypeError	TypeError	)

> More operators

Other built-in operators you might want to use:

```
//      # floor division: the whole-number quotient, rounded down  
%       # remainder: what remains after whole-number division  
**      # exponent: to the power of
```

> Types and output

What would we expect:

```
type(a / b)    float  
type(a // b)   int  
type(a % b)    int
```

> Using operators: calculating hours and minutes

```
total_minutes = 134
hours = total_minutes // 60
minutes = total_minutes % 60
print(hours, ":", minutes)
```

2 : 15

> Compound operators

Handy shortcut:

```
no_students = 34  
new_to_enroll = 6  
  
no_students = no_students + 6
```

```
no_students = 34  
new_to_enroll = 6  
  
no_students += 6
```

> Compound operators

Handy shortcut:

```
x = x + y    →    x += y
x = x - y    →    x -= y
x = x * y    →    x *= y
```

You can do this with any numerical operator!

> Type conversion

We can sometimes force Python to change the type of a value:

`int()` converts to an integer

`float()` converts to a float

`str()` converts to a string

This only works when it makes sense.

e.g.,

`int("2") = 2`

`int("hello") → error`

! Be careful doing this

Sometimes this creates unexpected outputs and sometimes it leads to loss of data.

- Converting a *float* to an *int* always loses information (what comes after the decimal point)
 - Sometimes this is fine, but don't do it by mistake
- It *always* rounds down, so may not give you the output you expect.
 - e.g., `int(9.99) = 9`

> Dot (.) notation for applying methods

In Python, like many languages, we use dot notation for applying *methods*.

A method is an operation associated with an object – instructions for how to do something to an object.

For example:

```
message = "hello"  
new_message = message.upper()  
print(new_message)
```

Output = HELLO

> Methods

Like classes, methods can be built-in or user-defined

Common and simple ones are built-in

Methods are of the form:

```
method_name(arguments)
```

Very often methods have zero arguments, but you still need to include the brackets.

> Methods for string manipulation

There are many of these built in. Common ones include:

```
<string>.upper()      # convert to uppercase  
<string>.lower()      # convert to lowercase  
<string>.replace(<old>,<new>) # replace part of a string with something else  
<string>.strip()      # removes white space
```

> Methods for string manipulation

These methods do not *change* a string – they create a new string

```
message = "hello"  
message.upper()  
print(message)
```

Output = hello

```
message = "hello"  
new_message = message.upper()  
print(new_message)
```

Output = HELLO

> Formatting output

`print()` prints out one or more arguments with a space in between

e.g., `print("a", "b")` returns a b

We've also seen how you can use `+` to concatenate strings and then print them out as a single argument.

e.g., `print("a" + "b")` returns ab

> Formatting output

We can do more interesting things with a print statement by using *f-strings*.

F-strings can evaluate expressions inside them if they are put in {}.

```
name = "Amira"  
subject = "Biology"  
room = 204  
  
print(f"Hello, {name}! Your {subject} class is in room {room}.")
```

Output:

```
Hello, Amira! Your Biology class is in room 204.
```