COMP10001

Numerical Expressions

Expressions and Statements

Two important structural terms for code are statements and expressions. These form the structure of each and every program.

Two important structural terms for code are **statements** and **expressions**. These form the structure of each and every program.

A **statement** is any valid line of Python code. Think of it like a sentence in a book. The following are all statements:

```
a = 5
print('this is a statement')
9 + 11
```

An **expression** is, as a simplified definition, a statement which evaluates to some value.

All expressions are statements, but not all statements are expressions. Only statements which evaluate to some value are expressions. `(2 + 3) * (5 - 7)` is both a statement and an expression.

Data Types

Type	Description
`int`	For whole numbers such as: -3, -5, or 10
`float`	For real numbers such as: -3.0, 0.5, or 3.14159
`bool`	The Boolean type. For storing `True` and `False` (only those two values; Booleans allow for no grey areas!).
`str` (= `"string"`)	For chunks of text, eg: `"Hello, I study Python"`
`tuple`	For combinations of objects, eg: `(1, 2, 3)` or `(1.0, "hello", "frank")`

Type	Description
`list`	A more powerful way of storing lists of objects, eg: `[1, 3, 4]` or `[1.0, "hello", "frank"]`
`dict`	We will see this later ... maybe you can guess what it does. eg: `{"bob": 34, "frankenstein": 203}`

Note that you can check the data type of any object with the type function.

```
print(type("<class 'dict'>"))
```

Output:

```
<class 'str'>
```

Integers and Float

Expression	Result
`int (*,+,-) int`	`int`
`int / int`	`float`
`float (*,+,-,/) float`	`float`
`float (*,+,-,/) int`	`float`
`int (*,+,-,/) float`	`float`

Floating point numbers allow us to represent a large set of numbers. Since there are infinitely many numbers (even in the range of -1 to 1) and the float data type has only a fixed amount of computer storage space available, some numbers will inevitably have rounding errors.

```
number_a = 12_345_678_901_234_567_890
number_b = 0.1
number_c = number_a + number_b - number_a
print(number_c)
```

Output:

```
0.0
```

Type Conversions and Input

- Type casting is the process of converting a value from one data type to another.
- Casting can be achieved by using the type's name as a function. For example, `int()` converts the value inside brackets into an integer.
- The `input()` function always returns a string, so casting is a useful way to convert input text into your preferred type.

More Operators

- The `//` **integer division** or **floor division** operator performs division but returns the result as an `int` by truncating the fractional part.
- The `%` **modulo** operator returns the remainder when the first operand is divided by the second.
- The `**` **exponentiation** operator returns the value of the first operand raised to the power of the second.

Conditonals

Type Conversion and Booleans

```
print(bool(1))
print(bool(-2.0))
print(bool("False"))
```

Output:

```
True
True
True
```

```
print(bool(0))
print(bool(0.0))
print(bool(""))
print(bool(False))
```

Output:

```
False
False
False
False
```

For every type, the "default" value converts to `False`, and everything else converts to `True`. For numeric types, this is the zero value, for strings it is the empty string, and for Booleans (not that you would ever want to convert a boolean into itself) it is `False`.

Relational Operators

Python supports six relational operators, summarized in the following table:

Python	Meaning	Math Notation
<	less than	<
<=	less than or equal	≤
==	equal	=
!=	not equal	≠
>	greater than	>
>=	greater than or equal	≥

When applied, each of these relational operators evaluates to a Boolean.

A Common Mistake

```
a = 1 + 1
b = 1
print(a = b)
```

❌

In practice, the final line print(a = b) is an assignment, rather than a test for equality (which would be print(a == b). Since assignment does not evaluate to anything (i.e. isn't an expression) it cannot be printed and Python will show an error message.

String Comparisons

All six relational operators can also be applied to strings.

```
print('he' < 'hi')
print('Hell' >= 'Hello')
```

Output:

```
True
False
```

```
print('h' > 'H')
print('Z' < 'a')
```

Output:

```
True
False
```

```
print(ord('A'))
print(ord('a'))
print('a' > 'A')
```

Output:

```
65
97
True
```

Substrings

```
print('moo' in 'wooloomooloo')
print('wooloomooloo' in 'moo')
```

Output:

```
True
False
```

➡️ Not commutative

Logical Expressions

Logical expressions are expressions which evaluate to a `bool` value. For instance, an expression using relational operators like `3 < 5` is a logical expression. A logical expression could also be just a literal value `True` or `False`, with no operators at all: `True` evaluates to `True`, after all!

Logical Operators: Combining Truth

To combine tests which evaluate to Boolean variables, we use **logical operators**, which take a pre-determined number of Boolean inputs and return a single Boolean value.

Python has two **binary logical operators** that apply to two Boolean variables:

- `and`
- `or`

and one **unary logical operator** that applies to a single Boolean variable:

- `not`

The `not` operator negates the logical expression it is applied to: if the expression `exp` is `False`, then `not exp` is `True`, and vice versa.

```
exp = False
print(not exp)
```

The rules for `and` and `or` are summarized as follows:

A	B	A and B	A or B
False	False	False	False
True	False	False	True
False	True	False	True
True	True	True	True

That is, for `and` to evaluate to `True`, **both** logical expressions it applies to must be `True`; for `or`, on the other hand, if **either** (or both) of the logical expressions it applies to is `True`, the combination is also `True`.

Logical Operators and Precedence

Just as Python defined precedence over arithmetic operators,
it also defines precedence over **logical and relational operators**, as follows:

relational operators (including `in`) > `not` > `and` > `or`

That is, relational operators are evaluated first, followed by `not`, etc.

```
print(not 1 > 2 and 1 > 0 or "din" in "coding")
```

Output:

```
True
```

Boolean and Logical Operator Summary

1. **A way of expressing truth**, to store whether a given condition holds or not.
   - This is done with **Boolean** values which can only be either true or false.
   - In Python the `bool` type holds one boolean value, either `True` or `False`
   - Other types can be converted to `bool` with the `bool()` function.
     The number zero or an empty string will convert to `False`; any other value will convert to `True`.
2. **A way of testing for a given condition**, often based on comparison between values.
   - Python's six **relational operators** (`>=`, `>`, `==`, `!=`, `<`, `<=`) compare two values, evaluating to a boolean result.

- It's especially common to confuse equality `==` and assignment `=` operators, so watch out for this.
- `in` can be used to test whether a substring is contained inside a superstring.
- A **logical expression** is any expression which evaluates to a `bool` result.
- Python's three **logical operators** `and`, `or` and `not` combine boolean values, or the results of logical expressions.
- `and` and `or` are **binary** operators, combining two boolean values into one. `not` is a **unary** operator, negating one particular boolean value.
- Python defines **precedence** over logical and relational operators as follows (from highest to lowest):

  ```
  relational operators (including `in`) > not > and > or
  ```

  Brackets are advised to make the order explicit.
- It is possible to **chain relational operators** to test for more complex ranges.

Conditional Blocks

```
n = int(input("Enter an integer: "))
if 0 < n < 6:
    print('You entered a positive integer less than six.')
print('Try again with another integer!')
```

The first `print()` statement here is **indented**. This is how Python knows what code to run when the `if` condition evaluates to `True`.

The combined set of indented statements underneath the `if` line is termed a **block**.
The general form of an `if` statement is as follows:

```
if <condition>:
    <block of statements>
```

The `<condition>` is usually a logical expression, involving relational or logical operators.
The indented block of statements is only executed if the condition evaluates to `True`.
Note that this notion of "block" is very important in Python,
and will come up again in the context of both functions and iteration.

Choosing among Two Alternatives

`if-else` statement:

```
if <condition>:
    <first block of statements>
else:
    <alternative block of statements>
```

Sequences

The main sequence data types we will deal with in this course are:

- **strings**
- **lists**
- **tuples**

Strings as Sequences

Strings are **sequences** of characters. Python numbers the position of each character within a string, starting with the first character at position number 0, and proceeding sequentially from left to right.

```
s = "The number is 42"
print(s[0])
print(s[1])
```

Output:

```
T
h
```

The largest valid index is one less than the length of the string (remember that we start counting from 0). To find out the largest valid index in a string, Python has a built-in function for that purpose: the `len()` function. The `len()` function is used like this:

```
s = "The number is 42"
n = len(s)
print(n)
print(len("Hello"))
```

Output:

```
17
5
```

Note that `len()` works a bit like the `print()` function: you give it the string you want to measure the length of inside parentheses. However, unlike the `print()` function (which has no output) the output of `len()` is an `int` : the number of characters in the string.

> Beware that the empty string has no characters, not even at index 0.

Negative String Indexing

```
s = "The number is 42."
print(s[-1])
print(s[-2])
print(s[-3])
print(s[-17])
```

Output:

```
.
2
4
T
```

> Note the negative indexes are one-offset (i.e. start from -1) while the positive indexes are zero-offset (i.e. start from 0). Why do you think this is?

Accessing Substrings (Slicing)

```
s = "The number is 42."
print(s[1] + s[2] + s[3] + s[4] + s[5])
```

```
s = "The number is 42."
print(s[1:6])
```

Both will ouput `he nu`

More Slicing

```
s = "The number is 42."
print(s[:5])
print(s[5:])
print(s[:])
```

Output:

```
The n
umber is 42.
The number is 42.
```

Changing the Step Size and Direction

You can specify a third number in a slice (after a second colon) which indicates how much to step through the sequence by for each item included in the slice. By default, this number is 1, indicating that every element should be included. If, instead of every item you wanted every second one, you can use a step value of 2:

```
s = "abcdef"
print(s[::2])
```

Output:

```
ace
```

If the step is -1, it reverses the direction of the sequence you are slicing. Note that the direction of the indices must be changed also (the rightmost character is now the start index and the leftmost is the end index):

```
s = "abcdef"
print(s[2::-1])
print(s[2:0:-1])
print(s[-4:-6:-1])
```

Output:

```
cba
cb
cb
```

Extension to Lists

Lists are for storing sequences of possibly different objects.
A list is a sequence just like a string, so there are many similarities between them. Let's compare: while strings are a sequence of **characters**, lists are sequences of **anything**. While strings use `'` or `"` as a delimiter, lists use `[]` (with a comma used to separate each item). Just like strings, lists can contain any number of elements and can be indexed and sliced, forwards and backwards, using the same syntax we learned earlier.

```
my_list = ['hello', 'world']
my_index = my_list[1]
print("index:", my_index)
print("slice:", my_slice)
print("equality:", my_index == my_slice)
```

Output:

```
index: world
slice: ['world']
equality: False
```

Extension to Tuples

A `tuple` is defined similarly to a `list`, but with round brackets (or "parentheses") rather than square brackets. An empty tuple is a pair of round brackets with nothing between them. A tuple with one item is a pair of round brackets around a value followed by a comma. This is essentially the same

as a `list` in that it can contain any combination of objects, but it can't be changed after creation (it is **immutable**).

Nested Sequences

Elements of nested sequences can be accessed by indexing twice.

```
my_tuple= ('name', 3, ['a', 'nested', 'list'], 'age')
print(my_tuple[2])
print(my_tuple[2][1])
print(my_tuple[2][1][:4])
```

Output:

```
['a', 'nested', 'list']
nested
nest
```

Basic Functions and Methods

```
# Calculate the number of standard drinks
# contained in `volume` litres of alcoholic
# drink with `percentage` % of alcohol
def std_drinks(volume, percentage):
    return volume * percentage * 0.789

# We use the function to do the calculation
print(std_drinks(0.375, 13.5))
```

The function name is `std_drinks`. `volume` and `percentage` are called **parameters**. They are very much like variables as you have seen to date, and define the inputs to the function. They take the values you provide when you **call** (i.e. use) the function. The values themselves are called **arguments**. The function is called by placing a pair of brackets `()` after the function name, and putting the arguments inside them.

Anatomy of a Function Definition

The basic components of a function are:

1. the function name;
2. the parameters;
3. the body, where the actual computation associated with the function occurs;
4. some mechanism for returning a value and exiting the function.

Important Return Notes

When the `return` statement is executed in a function, the function stops immediately, and returns the value given. This means that anything in your function after the `return` statement will not be executed.

```
def a_function():
    print("Returning a number:")
    return 1
    print("Returning another number:")
    return 3

print(a_function())
```

Output:

```
Returning a number:
1
```

However, it can be useful to have many return statements, if they are each executed in different circumstances. For example, they could be in different branches of an `if..elif..else` statement.

The other important point about `return` is that there doesn't have to be a return value.

```
def print_twice(output):
    print(output)
    print(output)
    return

print_twice("Hello there")
```

Output: