Lecture 6: Review of data types, conditionals and iteration

Contents

DSCI 511
Python Programming for Data Science

Lecture learning objectives

- Create, describe and differentiate standard Python data types such as `int`, `float`, `string`, `list`, `dict`, `tuple`, etc.
- Explain the difference between mutable objects like a list and immutable objects like a tuple
- Assign, index, slice and subset values to and from tuples, lists, strings and dictionaries
- Write a conditional statement with `if`, `elif` and `else`
- Identify code blocks by levels of indentation
- Write for and while loops in Python
- Identify iterable data types which can be used in for loops
- Create a list, dictionary, or set using comprehension

Common built-in Python data types

English name	Type name	Type Category	Description	Example
integer	`int`	Numeric Type	positive/negative whole numbers	`42`
floating point number	`float`	Numeric Type	real number in decimal form	`3.14159`
boolean	`bool`	Boolean Values	true or false	`True`
string	`str`	Sequence Type	text	`"Can I have a cheezburger?"`
list	`list`	Sequence Type	a collection of objects - mutable & ordered	`['Ali', 'Xinyi', 'Miriam']`
tuple	`tuple`	Sequence Type	a collection of objects - immutable & ordered	`('Thursday', 6, 9, 2018)`
dictionary	`dict`	Mapping Type	mapping of key-value pairs - mutable & ordered	`{'name':'DSCI', 'code':511, 'credits':2}`
none	`None`	Null Object	represents no value	`None`

Numeric data types

- There are three distinct numeric types: **integers**, **floating point numbers**, and **complex numbers** (not covered here)
- We can determine the type of an object in Python using `type()`
- We can print the value of the object using `print()`

```
x = 42
```

```
type(x)
```

```
int
```

```
print(x)
```

```
42
```

> With IPython, we don't need to explicitly use `print()` to see the content of a variable:

```
x
```

```
42
```

```
pi = 3.14159
```

```
print(pi)
```

```
3.14159
```

```
type(pi)
```

```
float
```

Arithmetic operators

- The syntax for the common arithmetic operators are:

Operator	Description
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`**`	exponentiation
`//`	integer division / floor division
`%`	modulo

- Let's apply these operators to numeric types and observe the results:

Addition

The `+` operator can be used to add two numbers together:

```
x = 2
y = 3
z = x + y
print(z)  # Output: 5
```

```
5
```

Subtraction

The `-` operator can be used to subtract one number from another:

```
x = 5
y = 3
z = x - y
print(z)  # Output: 2
```

```
2
```

Multiplication

The `*` operator can be used to multiply two numbers together:

```
x = 2
y = 3
z = x * y
print(z)  # Output: 6
```

```
6
```

Division

The `/` operator can be used to divide one number by another:

```
x = 6
y = 3
z = x / y
print(z)  # Output: 2.0
```

```
2.0
```

Note that the result of division is a floating-point number, even if both operands are integers.

```
type(4/2)
```

```
float
```

Floor Division

The `//` operator can be used to perform floor division, which returns the largest integer that is less than or equal to the result of the division:

```
x = 7
y = 3
z = x // y
print(z)  # Output: 2
```

```
2
```

Modulo

The `%` operator can be used to compute the remainder of division:

```
x = 7
y = 3
z = x % y
print(z)  # Output: 1
```

```
1
```

Exponentiation

The `**` operator can be used to raise a number to a power:

```
x = 2
y = 3
z = x ** y
print(z)  # Output: 8
```

```
8
```

Floating-Point Arithmetic in Python

In Python, floating-point numbers are represented using a finite number of bits, which means that they can only approximate real numbers. This can lead to rounding errors when performing arithmetic operations with floating-point numbers.

For example, consider the following code:

```
x = 0.1
y = 0.2
z = x + y
print(z == 0.3)  # Output: False
```

```
False
```

> Whoops! What is this? Why is `0.1 + 0.2` not exactly equal to `0.3`? Read more about it here

```
print(x)
print(y)
print(z)
```

```
0.1
0.2
0.30000000000000004
```

In this code, we would expect `z` to be equal to `0.3`, since `0.1 + 0.2` is equal to `0.3`. However, the output of the code is `False`.

This is because `0.1` and `0.2` cannot be represented exactly as floating-point numbers. When we add them together, the result is a floating-point number that is close to, but not exactly equal to, `0.3`. When we compare this result to `0.3`, the comparison returns `False`.

To avoid rounding errors when working with floating-point numbers, it is important to be aware of the limitations of floating-point arithmetic and to use appropriate methods for comparing floating-point numbers. One common method is

to use a tolerance value, which specifies how close two floating-point numbers must be to be considered equal.

For example, we could us np.isclose() function with a tolerance value of `1e-9`:

```
import numpy as np

x = 0.1
y = 0.2
z = x + y
tolerance = 1e-9
np.isclose(z, 0.3, atol=tolerance)
```

```
np.True_
```

Default order of operations:

1. Parentheses
2. Exponentiation
3. Multiplication
4. Division
5. Addition
6. Subtraction.

- What is the result of the following calculation?

```
34 + 10 / 2 * (2 - 3)
```

```
29.0
```

In this code, we use the `np.isclose()` function to compare `z` and `0.3` with a tolerance value of `1e-9`. If the absolute difference between `z` and `0.3` is less than or equal to the tolerance value, the comparison returns `True`.

By using appropriate methods for comparing floating-point numbers, we can avoid many of the rounding errors that can occur when working with floating-point arithmetic in Python.

None

- `NoneType` is its own type in Python.
- It only has one possible value, `None` - it represents an object with no value

```
x = None
```

```
print(x)
```

```
None
```

```
type(x)
```

```
NoneType
```

- You may have seen similar things in other languages, like `null` in Java, etc.

> Note: `None` is related to `NaN`, but there are some key differences. In particular Stands for “Not a Number” and is used specifically in numerical contexts to represent undefined or unrepresentable numbers, such as the result of

> `0/0`. It is a floating-point value and belongs to the float type (from the `math` module or `numpy` library). Other interesting things to note are the differnece in comparisons; `None == None` returns `True`, while `numpy.nan == math.nan == math.nan .nan` returns `False`.

Comparison operators

- Compare objects using comparison operators, with a Boolean result:

Operator	Description
`x == y`	is `x` equal to `y`?
`x != y`	is `x` not equal to `y`?
`x > y`	is `x` greater than `y`?
`x >= y`	is `x` greater than or equal to `y`?
`x < y`	is `x` less than `y`?
`x <= y`	is `x` less than or equal to `y`?
`x is y`	is `x` the same object as `y`?

```
2 < 3
```

```
True
```

```
"Deep learning" == "Solve all the world's problems"
```

```
False
```

```
2 != "2"
```

```
True
```

`is` versus `==`

```
2 is 2
```

```
<>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
/var/folders/2w/xsy5y_ms0t74ssbfqt_m0tr80000gp/T/ipykernel_20821/564785565.py:1: SyntaxWarning: "is" wi
  2 is 2
```

```
True
```

```
2 == 2.0
```

```
True
```

```

```

```
2 is 2.0
```

```
<>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
/var/folders/2w/xsy5y_ms0t74ssbfqt_m0tr80000gp/T/ipykernel_20821/700261715.py:1: SyntaxWarning: "is" wi
  2 is 2.0
```

```
False
```

In Python, the `is` operator and the `==` operator are used to compare objects, but they have different meanings.

The `is` operator checks if two variables refer to the same object in memory.

On the other hand, the `==` operator checks if two variables have the same value.

In the case of `2 == 2.0`, the `==` operator checks if the values of `2` and `2.0` are the same. Since `2` and `2.0` have the same numerical value, the comparison returns `True`.

However, if we use the `is` operator to compare `2` and `2.0`, the comparison will return `False`. This is because `2` and `2.0` are not the same object in memory, even though they have the same value.

```
# check the location of an object in memory
id(2)
```

```
4404512696
```

```

```

```
# check the location of an object in memory
id(2.0)
```

```
4454725968
```

Boolean

- The Boolean (`bool`) type has two values: `True` and `False`.
- They are pretty self-explanatory and are constant objects in Python (they always exist).

```
the_truth = True
```

```
print(the_truth)
```

```
True
```

```
type(the_truth)
```

```
bool
```

```
lies = False
```

```
print(lies)
```

```
False
```

```
type(lies)
```

```
bool
```

Boolean operators

- Evaluates to either `True` or `False`:

Operator	Description
`x and y`	are `x` and `y` both True?
`x or y`	is at least one of `x` and `y` True?
`not x`	is `x` False?

```
True and True
```

```
True
```

```
True and False
```

```
False
```

```
True or False
```

```
True
```

```
False or False
```

```
False
```

```
('Python 2' != 'Python 3') and (2 <= 3)
```

```
True
```

```
True
```

```
True
```

```
not True
```

```
False
```

```
not not True
```

```
True
```

Casting/Conversion

- Sometimes (but pretty rarely) we need to explicitly **cast** or **convert** a value from one type to another.
- Python tries to do something reasonable, or throws an error if it has no ideas.

```
x = 5.0
type(x)
```

```
float
```

```
x = int(5.0)
x
```

```
5
```

```
type(x)
```

```
int
```

```
type(5)
```

```
int
```

```
float(5)
```

```
5.0
```

```
x = str(5.0)
x
```

```
'5.0'
```

```
type(x)
```

```
str
```

```
str(5.0) == 5.0
```

```
False
```

```
int(5.3)
```

```
5
```

```
float('hello')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[54], line 1
----> 1 float('hello')

ValueError: could not convert string to float: 'hello'
```

> Does `a // b` (floor division) do the same thing as `int(a / b)`?

- `int()` merely takes away what comes after the decimal point, whereas `//` rounds down:

```
5 / 3
```

```
1.6666666666666667
```

```
int(5 / 3)
```

```
1
```

```
int(-5 / 3)
```

```
-1
```

```
5 // 3
```

```
1
```

```
-5 // 3
```

```
-2
```

Strings

Strings are a core data type in Python that represent sequences of characters. They are used to store text-based data, such as words, sentences, and more complex textual information.

Creating Strings

Strings can be created using single quotes (`'`) or double quotes (`"`).

```
single_quoted = 'Hello, World!'
double_quoted = "Python Programming"
```

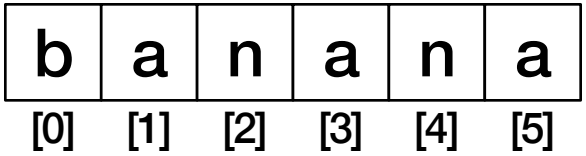
- There's another way to prevent Python from interpreting `'` and `"` as string delimiters
- We can escape special characters—such as a `'`, which has a special meaning in Python—using `\`:

```
print('It\'s a rainy day.')
```

```
It's a rainy day.
```

String Indexing and Slicing

Strings can be accessed using indexes, allowing you to retrieve individual characters or subsequences.



Credit: Python 4 everyone

```
my_string = "banana"
print(my_string[0])  # Output: 'P'
print(my_string[1:4])  # Output: 'yth'
```

```
b
ana
```

String Concatenation

Strings can be concatenated using the `+` operator, allowing you to combine multiple strings.

```
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name  # Output: 'John Doe'
```

String Methods and Utilities

1. Length

The `len()` function returns the length of a string, i.e., the number of characters it contains.

```
text = "Hello, World!"
length = len(text)  # Output: 13
```

2. Case Conversion

You can convert strings to lowercase or uppercase using the `lower()` and `upper()` methods.

```
text = "Python Programming"
lower_text = text.lower()  # Output: 'python programming'
upper_text = text.upper()  # Output: 'PYTHON PROGRAMMING'
```

3. Stripping Whitespace

You can remove leading and trailing whitespace from strings using `strip()`, `lstrip()`, and `rstrip()`.

```
text = "   Spaces   "
stripped_text = text.strip()  # Output: 'Spaces'
stripped_text
```

```
'Spaces'
```

4. Finding Substrings

The `find()` and `index()` methods help you locate substrings within a string.

```
text = "Python Programming"
index = text.index("Prog")  # Output: 7
index
```

```
7
```

5. Replacing Substrings

The `replace()` method allows you to replace occurrences of a substring with another substring.

```
text = "Hello, World!"
new_text = text.replace("Hello", "Hi")  # Output: 'Hi, World!'
new_text
```

```
'Hi, World!'
```

6. Splitting and Joining

The `split()` method splits a string into a list of substrings based on a delimiter. The `join()` method joins a list of strings using a delimiter.

```
text = "This is Canada"
text.split()
```

```
['This', 'is', 'Canada']
```

```
text = "This-is-Canada"
text.split('-')
```

```
['This', 'is', 'Canada']
```

String formatting

- Python has ways of creating strings by “filling in the blanks” and formatting them nicely.

- This is helpful for when you want to print statements that include variables or statements.
- There are a few ways of doing this. See here and here for some discussion.
- I use and recommend f-strings which were introduced in Python 3.6. See format code options here.

```
name = 'Newborn Baby'
age = 4 / 12
day = 10
month = 6
year = 2021
template_new = f'Hello, my name is {name}. I am {age:.2f} years old. I was born on {day}/{month:02}/{ye

template_new
```

```
'Hello, my name is Newborn Baby. I am 0.33 years old. I was born on 10/06/2021.'
```

`print()` behavior

- The `print()` function puts spaces between the printed objects by default and also moves to the next line.
- But we also have the option of changing the default behaviour:

python print(*objects, sep=' ', end='\n', file=sys.stdout)

```
# default behavior:
a, b, c = True, 30, 'UBC'
print(a, b)
print(c)
```

```
True 30
UBC
```

We can change the separation and end characters:

```
print(a, b, sep='_', end=' ')
print(c)
```

```
True_30 UBC
```

Escape characters

- With the `print()`, you've probably noticed the use of a special combination such as `\n` for the `end` argument.
- This is called an **escape character**:

Escape character	What it prints
`\\`	Backslash `\`
`\'`	Single quote `'`
`\"`	Double quote `\`
`\n`	New line
`\t`	Tab space

```
a, b, c = 1.5, 30, 'MDS'
print('The variable value is: \n\t\t\t{a}')
print(f'The variable value is: \n\t\t\t{a}')
```

```
The variable value is:
                        {a}
The variable value is:
                        1.5
```

- We can precede strings with the letter `r` (`r` for raw text) just as we did with `f` (formatted strings). The `r` tells Python to treat everything as **raw text** and not look for escape characters.

```
print(r'The variable value is: \n\t\t\t{a}')
```

```
The variable value is: \n\t\t\t{a}
```

- Remember this example from before?

```
quote = 'Donald Knuth: "Premature optimization is the root of all evil."'
print(quote)
```

```
Donald Knuth: "Premature optimization is the root of all evil."
```

- With escape characters, we no longer need to use both types of quotes. Literal quotes can simply be escaped:

```
quote = "Donald Knuth: \"Premature optimization is the root of all evil.\""
print(quote)
```

```
Donald Knuth: "Premature optimization is the root of all evil."
```

Lists

Lists are one of the most versatile and widely used data structures in Python. They allow you to store collections of items, such as numbers, strings, or other objects, in a single variable. Lists are mutable, which means you can modify their contents after creation.

Creating Lists

Lists are created by enclosing items in square brackets `[]`, separated by commas.

```
numbers = [1, 2, 3, 4, 5]
fruits = ["apple", "banana", "orange"]
mixed_list = [10, "Hello", 3.14]
```

Accessing List Elements

List elements can be accessed using indexing, starting from 0.

```
fruits = ["apple", "banana", "orange"]
print(fruits[0])  # Output: 'apple'
```

```
apple
```

Changing List Elements

Lists are mutable, which means you can change their elements after creation.

```
fruits = ["apple", "banana", "orange"]
fruits[1] = "grape"
print(fruits)  # Output: ['apple', 'grape', 'orange']
```

```
['apple', 'grape', 'orange']
```

List Slicing

You can use slicing to extract a portion of a list.

```
numbers = [1, 2, 3, 4, 5]
sliced_numbers = numbers[1:4]  # Output: [2, 3, 4]
```

List Methods and Utilities

1. Length

The `len()` function returns the number of items in a list.

```
numbers = [1, 2, 3, 4, 5]
length = len(numbers)  # Output: 5
```

2. Adding Elements

The `append()` method adds an element to the end of a list. The `insert()` method inserts an element at a specified position.

```
fruits = ["apple", "banana"]
fruits.append("orange")  # Now, fruits = ["apple", "banana", "orange"]
fruits.insert(1, "grape")  # Now, fruits = ["apple", "grape", "banana", "orange"]
```

3. Removing Elements

The `remove()` method removes the first occurrence of a specified value. The `pop()` method removes and returns an element at a given index.

```
fruits = ["apple", "banana", "orange"]
fruits.remove("banana")  # Now, fruits = ["apple", "orange"]
removed_fruit = fruits.pop(1)  # Now, fruits = ["apple"], removed_fruit = "orange"
```

4. Sorting

The `sort()` method sorts a list in place. The `sorted()` function returns a new sorted list.

```
numbers = [3, 1, 4, 2, 5]
numbers.sort()  # Now, numbers = [1, 2, 3, 4, 5]
sorted_numbers = sorted(numbers)  # sorted_numbers = [1, 2, 3, 4, 5]
```

5. Reversing

The `reverse()` method reverses the order of elements in a list.

```
numbers = [1, 2, 3, 4, 5]
numbers.reverse()  # Now, numbers = [5, 4, 3, 2, 1]
```

6. Membership Testing

You can use the `in` keyword to check if an element is present in a list.

```
fruits = ["apple", "banana", "orange"]
print("banana" in fruits)  # Output: True
```

```
True
```

Dictionaries

Dictionaries are powerful and versatile data structures in Python that allow you to store data in key-value pairs. Unlike lists that use numeric indices, dictionaries use unique keys to access their values. Dictionaries are unordered and mutable, providing efficient lookups and modification of data.

Creating Dictionaries

Dictionaries are created using curly braces `{}` and contain key-value pairs separated by colons `:`.

```
student = {
    "name": "John",
    "age": 20,
    "major": "Computer Science"
}
```

Accessing and Modifying Values

Values in dictionaries are accessed using their corresponding keys.

```
student = {
    "name": "John",
    "age": 20,
    "major": "Computer Science"
}
print(student["name"])  # Output: 'John'
```

```
John
```

The `get()` method provides a way to access dictionary values using a key, with an optional default value if the key is not found.

```
name = student.get("name")
print("Name:", name)  # Output: Name: John
```

```
Name: John
```

```
# Using get() with a default value
gender = student.get("gender", "Unknown")
print("Gender:", gender)  # Output: Gender: Unknown
```

```
Gender: Unknown
```

Dictionary Methods and Utilities

1. Adding and Updating Items

You can add or update items in a dictionary using their keys.

```
student["GPA"] = 3.8  # Adding a new item
student["age"] = 22  # Updating an existing item
```

2. Removing Items

You can use the `del` statement or the `pop()` method to remove items from a dictionary.

```
del student["major"]  # Remove using del
gpa = student.pop("GPA")  # Remove using pop
```

3. Length

The `len()` function returns the number of key-value pairs in a dictionary.

```
num_items = len(student)  # Output: 2 (if 'major' is removed)
num_items
```

```
2
```

4. Checking Membership

You can use the `in` keyword to check if a key is present in a dictionary.

```
if "name" in student:
    print("Name is present")
```

```
Name is present
```

5. Getting Keys and Values

You can get lists of keys, values, or key-value pairs using these methods.

```
keys = student.keys() # get all keys
values = student.values() # get all values
items = student.items() # get all items

print(keys, values, items, sep='\n')
```

```
dict_keys(['name', 'age'])
dict_values(['John', 22])
dict_items([('name', 'John'), ('age', 22)])
```

6. Clearing and Copying

The `clear()` method removes all items from a dictionary, and `copy()` creates a shallow copy of the dictionary.

```
student.clear()  # Removes all items
student_copy = student.copy()  # Creates a copy
```

Tuples

A tuple is an ordered, immutable collection of elements in Python. Tuples are similar to lists, but they cannot be modified once they are created. Tuples are often used to represent fixed collections of related values.

Here's an example of creating a tuple in Python:

```
my_tuple = (1, 2, 3)
```

In this example, we create a tuple `my_tuple` with the values `1`, `2`, and `3`.

We can access individual elements of a tuple using indexing, just like with lists:

```
print(my_tuple[0])  # Output: 1
print(my_tuple[1])  # Output: 2
print(my_tuple[2])  # Output: 3
```

```
1
2
3
```

We can also use slicing to access a range of elements in a tuple:

```
print(my_tuple[1:3])  # Output: (2, 3)
```

```
(2, 3)
```

Note that since tuples are immutable, we cannot modify individual elements of a tuple:

```
my_tuple[0] = 4  # Raises a TypeError
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[101], line 1
----> 1 my_tuple[0] = 4  # Raises a TypeError

TypeError: 'tuple' object does not support item assignment
```

This code raises a `TypeError`, because we are trying to modify an element of a tuple, which is not allowed.

Sets

A set is an unordered collection of unique elements in Python. Sets are similar to lists and tuples, but they cannot contain duplicate elements. Sets are often used to perform mathematical operations such as union, intersection, and difference.

Here's an example of creating a set in Python:

```
my_set = {1, 2, 3}
```

In this example, we create a set `my_set` with the values `1`, `2`, and `3`.

We can add elements to a set using the `add()` method:

```
my_set.add(4)
```

We can remove elements from a set using the `remove()` method:

```
my_set.remove(3)
```

We can perform mathematical operations on sets using methods such as `union()`, `intersection()`, and `difference()`:

```
set1 = {1, 2, 3}
set2 = {2, 3, 4}

print(set1.union(set2))  # Output: {1, 2, 3, 4}
print(set1.intersection(set2))  # Output: {2, 3}
print(set1.difference(set2))  # Output: {1}
```

```
{1, 2, 3, 4}
{2, 3}
{1}
```

Note that since sets are unordered, we cannot access individual elements of a set using indexing or slicing.

Comparison between lists, tuples, sets, dict

Feature	List	Tuple	Dictionary (dict)	Set
Mutable	Yes	No	Yes	Yes
Ordered	Yes	Yes	Yes (as of Python 3.7)	No
Indexing	By index (0-based)	By index (0-based)	By key	No
Duplicates	Allowed	Allowed	Keys must be unique	Duplicates not allowed
Modification	Can change elements	Cannot change elements	Values can be updated	Elements can be added/removed
Syntax	Square brackets `[]`	Parentheses `()`	Curly braces `{}`	Curly braces `{}`
Use Case	When order matters	When data should not change	Mapping keys to values	For unique values and set operations
Example	`[1, 2, 'three']`	`(1, 2, 'three')`	`{'name': 'Alice', 'age': 30}`	`{'apple', 'banana', 'cherry'}`

Conditionals

- Conditional statements allow us to write programs where only certain blocks of code are executed depending on the state of the program.
- Check out the Python documentation and Think Python (Chapter 5) for more information about conditional execution.- Check out the Python documentation and Think Python (Chapter 5) for more information about conditional execution.

The `if` Statement

The `if` statement is used to execute a block of code if a certain condition is true. Here's the basic syntax of an `if` statement in Python:

```
if condition:
    # code to execute if condition is true
```

In this syntax, `condition` is a Boolean expression that evaluates to either `True` or `False`.

If `condition` is `True`, the code block indented under the `if` statement will be executed.

If `condition` is `False`, the code block will be skipped.

Here's an example of using an `if` statement in Python:

```
x = 5

if x > 0:
    print("x is positive")
```

```
x is positive
```

The `if-else` Statement

The `if-else` statement is used to execute one block of code if a certain condition is true, and another block of code if the condition is false. Here's the basic syntax of an `if-else` statement in Python:

```
if condition:
    # code to execute if condition is true
else:
    # code to execute if condition is false
```

```
x = -5

if x > 0:
    print("x is positive")
else:
    print("x is not positive")
```

```
x is not positive
```

If `condition` is `True`, the code block indented under the `if` statement will be executed.

If `condition` is `False`, the code block indented under the `else` statement will be executed.

Here's an example of using an `if-else` statement in Python:

The `if-elif-else` Statement

The `if-elif-else` statement is used to execute different blocks of code depending on multiple conditions. Here's the basic syntax of an `if-elif-else` statement in Python:

```
if condition1:
    # code to execute if condition1 is true
elif condition2:
    # code to execute if condition2 is true
else:
    # code to execute if all conditions are false
```

Here's an example of using an `if-elif-else` statement in Python:

```
x = 0

if x > 0:
    print("x is positive")
elif x < 0:
    print("x is negative")
else:
    print("x is zero")
```

```
x is zero
```

Nested Conditionals

Nested conditionals are conditional statements that are nested inside other conditional statements. Here's an example of a nested `if-else` statement in Python:

```
x = 5
y = 10

if x > 0:
    if y > 0:
        print("x and y are both positive")
    else:
        print("x is positive but y is not")
elif x < 0:
    if y > 0:
        print("x is negative but y is positive")
    else:
        print("x and y are both negative")
else:
    print("x and y are both zero")
```

```
x and y are both positive
```

- The main points to notice:
  - **Indentation (by 4 empty space) defines code blocks**
  - Use keywords `if`, `elif` and `else`
  - The colon `:` ends each conditional expression
  - In an `if` statement, the first block whose conditional statement returns `True` is executed and the program exits the `if` block
  - `if` statements don't necessarily need `elif` or `else`
  - `elif` lets us check several conditions
  - `else` lets us evaluate a default block if all other conditions are `False`
  - the end of the entire `if` statement is where the indentation returns to the same level as the first `if` keyword

Multi-way puzzle

What do you expect the output of the following code will be?

```
x = 5
y = -10

if x > 0:
    if y > 0:
        if x > y:
            print("x is greater than y")
        elif x < y:
            print("x is less than y")
        else:
            print("x is equal to y")
    else:
        print("y is not positive")
else:
    print("x is not positive")
```

```
y is not positive
```

Truth Value Testing

In Python, every value has a truth value, which is either `True` or `False`. Here are some examples of truth value testing in Python:

```
bool(0)  # False
bool(1)  # True
bool("")  # False
bool("hello")  # True
bool([])  # False
bool([1, 2, 3])  # True
```

```
True
```

In Python, the following values are considered `False`:

- `False`
- `None`
- `0`
- `0.0`
- `""`
- `[]`
- `()`
- `{}`

All other values are considered `True`.

Truth value testing is often used in conditional statements to check if a certain value is `True` or `False`. For example:

```
x = 5

if x:
    print("x is not zero")
else:
    print("x is zero")
```

```
x is not zero
```

Inline if/else

- We can write simple `if` statements "inline", i.e., in a single line

```
words = ["the", "list", "of", "words"]

x = "long list" if len(words) > 10 else "short list"
x
```

```
'short list'
```

Short-circuiting

- Python supports short-circuiting
- This is the automatic stopping of the execution of boolean operation if the truth value of expression has already been determined

```
fake_variable  # not defined
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[114], line 1
----> 1 fake_variable  # not defined

NameError: name 'fake_variable' is not defined
```

```
True or fake_variable
```

```
True
```

```
True and fake_variable
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[116], line 1
----> 1 True and fake_variable

NameError: name 'fake_variable' is not defined
```

```
False and fake_variable
```

```
False
```

Expression	Result	Detail
A or B	If A is `True` then A else B	B only executed if A is `False`
A and B	If A is `False` then A else B	B only executed if A is `True`

`for` loops

In programming, a loop is a sequence of instructions that is executed repeatedly until a certain condition is met. One type of loop is the **for loop**, which is used to iterate over a sequence of values.

```
for variable in sequence:
    # code to be executed
```

Below we create a list containing the integers 1, 2, 3 and 4. And we create a temporary variable, `n` to hold an integer for each iteration of the loop. The loop will perform as many iteration as there are items in our list.

The first time through the loop, `n` will be 1, the second time through the loop, `n` will be 2, the third time through the loop, `n` will be 3, and the fourth (and last) time through the loop, `n` will be 4.

Because the loop has 4 iterations, we will get 4 print statements. Each using one of the values from the list.

```
for n in [1, 2, 3, 4]:
    print(f"The number is {n} and its square is {n**2}")

print("I'm outside the loop!")
```

```
The number is 1 and its square is 1
The number is 2 and its square is 4
The number is 3 and its square is 9
The number is 4 and its square is 16
I'm outside the loop!
```

The main points to notice:

- Keyword `for` begins the loop. Colon `:` ends the first line of the loop.
- The indented block of code is executed for each value in the list (hence the name “for” loops)
- The loop ends after the variable `n` has taken all the values in the list
- We can iterate over any kind of **“iterable”**: `list`, `tuple`, `range`, `set`, `string`.

- An iterable is really just any object with a sequence of values that can be looped over. In this case, we are iterating over the values in a list.

```
word = "Python"
for letter in word:
    print("Gimme a " + letter + "!")

print(f"What's that spell?!! {word}!")
```

```
Gimme a P!
Gimme a y!
Gimme a t!
Gimme a h!
Gimme a o!
Gimme a n!
What's that spell?!! Python!
```

- A very common pattern is to use `for` with the `range` object.
- `range` gives you a sequence of integers up to some value (non-inclusive of the end-value) and is typically used for looping.

```
range(10)
```

```
range(0, 10)
```

```
list(range(10))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
for i in range(10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

- We can also specify a start value and a step value with `range`:

```
for i in range(1, 101, 10):
    print(i)
```

```
1
11
21
31
41
51
61
71
81
91
```

- We can write a loop inside another loop to iterate over multiple dimensions of data. Consider the following loop as enumerating the coordinates in a 3 by 3 grid of points.

```
list_1 = [0, 1, 2]
list_2 = ["a", "b", "c"]
```

```
for x in list_1:
    for y in list_2:
        print((x, y))
```

```
(0, 'a')
(0, 'b')
(0, 'c')
(1, 'a')
(1, 'b')
(1, 'c')
(2, 'a')
(2, 'b')
(2, 'c')
```

Indices can be useful when you want to iterate over to different objects:

```
for i in range(3):
    print(list_1[i], list_2[i])
```

```
0 a
1 b
2 c
```

- There are many clever ways of doing these kinds of things in Python
- `zip()` returns a zip object which is an iterable of tuples

```

```

```
for i in zip(list_1, list_2):
    print(i)
```

```
(0, 'a')
(1, 'b')
(2, 'c')
```

- We can "unpack" this tuples directly in the `for` loop:

```
for i, j in zip(list_1, list_2):
    print(i, j)
```

```
0 a
1 b
2 c
```

- `enumerate()` adds a counter to an iterable. It creates an iterable that contains tuples of the form `(index, element)` for each element in the list.

```
for i in enumerate(list_2):
    print(i)
```

```
(0, 'a')
(1, 'b')
(2, 'c')
```

- It is also possible to extract the counter and the value:

```
list_2
```

```
['a', 'b', 'c']
```

```
for n, i in enumerate(list_2):
    print(f"index {n}, value {i}")
```

```
index 0, value a
index 1, value b
index 2, value c
```

- We can loop through key-value pairs of a dictionary using `.items()`
- The general syntax is `for key, value in dictionary.items()`

```
courses = {521: "awesome 🤗",
           551: "riveting 🤯",
           523: "a party time 🥳",
           511: "a slithering good time 🐍"}
```

```
for course_num, description in courses.items():
    print(f"DSCI {course_num}, is {description}")
```

```
DSCI 521, is awesome 🤗
DSCI 551, is riveting 🤯
DSCI 523, is a party time 🥳
DSCI 511, is a slithering good time 🐍
```

- We can even use `enumerate()` to do more complex un-packing:

```
for n, (course_num, description) in enumerate(courses.items()):
    print(f"Item {n}: DSCI {course_num}, is {description}")
```

```
Item 0: DSCI 521, is awesome 🤗
Item 1: DSCI 551, is riveting 🤯
Item 2: DSCI 523, is a party time 🥳
Item 3: DSCI 511, is a slithering good time 🐍
```

Challenge: Pairwise Differences

Write a loop that takes a list of integers as input and returns a new list that contains the absolute differences between each pair of adjacent elements in the input list.

For example, if the input list is `[1, 3, 5, 7]`, the output list should be `[2, 2, 2]`, which represents the absolute differences between adjacent elements: `abs(3-1)`, `abs(5-3)`, and `abs(7-5)`.

`while` loops

- We can also use a **`while`** loop to execute a block of code until a condition becomes `False`.
- Beware! If the conditional expression is always `True`, then you've got an infinite loop!

> (Use the "Stop" button in the toolbar above, or `Ctrl+C` in the terminal, to kill the program if you get an infinite loop.)

Here is the structure of a `while` loop:

```
while some_condition:
    # do these things
```

Below we show an example of a simple `while` loop to get us started:

```
n = 5
while n > 0:
    print(n)
    n -= 1

print("Lift off!")
```

```
5
4
3
2
1
Lift off!
```

- We can read the `while` statement above as if it were English.
- It means,

> *While n is greater than 0, display the value of n and then decrement n by 1. When you get to 0, display "Lift off!"*

- But for some loops, it's hard to tell when, or if, they will stop!

Coding challenge: the Collatz conjecture

Pick a starting positive integer $ n $. Next terms are obtained as follows: if the previous term is even, the next term is one half of the previous term. If the previous term is odd, the next term is 3 times the previous term plus 1

The conjecture states that no matter what positive integer $ n $ we start with, the sequence will always eventually reach 1

```
n = 5

while n != 1:
    print(n)
    if n % 2 == 0:  # n is even
        n = n // 2
    else:  # n is odd
        n = n * 3 + 1

print(n)
```

```
5
16
8
4
2
1
```

- In some cases, you may want to force a `while` loop to stop based on some criteria, using the `break` keyword

```
n = 123
i = 0

while n != 1:
    print(n)
    if n % 2 == 0:  # n is even
        n = n // 2
    else:  # n is odd
        n = n * 3 + 1
    i += 1
    if i == 10:
        print(f"Ugh, too many iterations!")
        break
```

```
123
370
185
556
278
139
418
209
628
314
Ugh, too many iterations!
```

- The `continue` keyword is similar to `break` but won't stop the loop
- Instead, `continue` ignores what comes after it, and goes to the next iteration.

```
n = 10
while n > 0:
    if n % 2 != 0:  # n is odd
        n -= 1
        continue
        print('Can anyone hear me?')  # this line is never executed
    print(n)
    n -= 1

print("Lift off!")
```

```
10
8
6
4
2
Lift off!
```

Comprehensions

Iteration in comprehensions

Comprehensions allow us to build lists/tuples/sets/dictionaries in one convenient, compact line of code. Below is a standard for loop you might use to iterate over an iterable and create a list

```
subliminal = ['Toby', 'ingests', 'many', 'eggs', 'to', 'outrun',
              'large', 'eagles', 'after', 'running', 'near', '!']

first_letters = []

for word in subliminal:
    first_letters.append(word[0])

print(first_letters)
```

```
['T', 'i', 'm', 'e', 't', 'o', 'l', 'e', 'a', 'r', 'n', '!']
```

A list comprehension allows us to do this in one line

```
letters = [word[0] for word in subliminal]  # list comprehension
letters
```

```
['T', 'i', 'm', 'e', 't', 'o', 'l', 'e', 'a', 'r', 'n', '!']
```

Similar to `for` loops, we can choose to use indexing instead of iterating directly over the items of the iterable. Here's the example above using indexing:

```
[subliminal[i][0] for i in range(len(subliminal))]
```

```
['T', 'i', 'm', 'e', 't', 'o', 'l', 'e', 'a', 'r', 'n', '!']
```

We can do more complex loops with comprehensions, including multiple iterations. Below we perform a comprehension that is equivalent to a nested for loop, where the outer loop is `for i in range(3)` and the inner loop is `for j in range(4)`:

```
[(i, j) for i in range(3) for j in range(4)]
```

```
[(0, 0),
 (0, 1),
 (0, 2),
 (0, 3),
 (1, 0),
 (1, 1),
 (1, 2),
 (1, 3),
 (2, 0),
 (2, 1),
 (2, 2),
 (2, 3)]
```

To convince you of this, here's the equivalent code as a nested for loop:

```

```

```
my_list = []
for i in range(3):
    for j in range(4):
        my_list.append((i, j))

my_list
```

```
[(0, 0),
 (0, 1),
 (0, 2),
 (0, 3),
 (1, 0),
 (1, 1),
 (1, 2),
 (1, 3),
 (2, 0),
 (2, 1),
 (2, 2),
 (2, 3)]
```

Conditionals in comprehensions

Condition the iterator to select even numbers only:

```
[i for i in range(11) if i % 2 == 0]
```

```
[0, 2, 4, 6, 8, 10]
```

Condition the iterator to multiply `i` by `-1` if `i` is even:

```
[-i if i % 2 == 0 else i for i in range(11)]
```

```
[0, 1, -2, 3, -4, 5, -6, 7, -8, 9, -10]
```

Note that:

- if you only want to keep certain elements, put the condition at the end. **You can't have an `else` statement in this case.**

- If you need all elements in the list, put the condition at the beginning. **Having an `else` statement is mandatory in this case**

Comprehensions and data types

We are not limited to lists for comprehensions. We can make a comprehension with several iterables, including lists, tuples, sets and dictionaries.

Below we demonstrate a set comprehension that returns the set of first characters from a list of words:

```
words = ['hello', 'goodbye', 'the', 'antidisestablishmentarianism']
y = {word[-1] for word in words}  # set comprehension
y
```

```
{'e', 'm', 'o'}
```

We see only 3 elements because a set contains only unique items and there would have been two “e”s.

Below we demonstrate a dictionary comprehension that iterates over a list, and returns a dictionary where the keys are the words, and the values are the number of characters in each word:

```
words = ['hello', 'goodbye', 'the', 'antidisestablishmentarianism']
word_lengths = {word: len(word) for word in words}  # dictionary comprehension
word_lengths
```

```
{'hello': 5, 'goodbye': 7, 'the': 3, 'antidisestablishmentarianism': 28}
```

Interestingly, there are not tuple comprehensions in Python. Instead we need to create a generator that we then convert to a tuple.

What the heck is a generator? It is a special type of function in Python that returns a lazy iterator (an iterator that is not immediately evaluated). Given that it is not immediately evaluated, these can be more memory efficient as well as time efficient for larger data sets. Generators can be iterated over later to yield their values. It is important to note however, that once a generator is iterated over, it cannot be re-used

Later in MDS we will get more into generators, for now we will only demonstrate how we can write a generator expression to get a tuple.

Generator comprehensions use `(` notation (which differs from the `[` and `{` syntax used by lists, sets and dictionaries). And the iteration setup inside the generator comprehension is the same as that of other comprehensions.

Going back to our first example, where we have the `words` list and we want to get the first letter from each word, if we use `(` brackets, we obtain a generator object:

```
subliminal
```

```
['Toby',
 'ingests',
 'many',
 'eggs',
 'to',
 'outrun',
 'large',
 'eagles',
 'after',
 'running',
 'near',
 '!']
```

```
(word[0] for word in subliminal)
```

```
<generator object <genexpr> at 0x1099de0c0>
```

We can then get the values back from the generator object by applying a function like `list` or `tuple`, or `set` to evaluate it:

```
list(word[0] for word in subliminal)
```

```
['T', 'i', 'm', 'e', 't', 'o', 'l', 'e', 'a', 'r', 'n', '!']
```

```
tuple(word[0] for word in subliminal)
```

```
('T', 'i', 'm', 'e', 't', 'o', 'l', 'e', 'a', 'r', 'n', '!')
```

```
set(word[0] for word in subliminal)
```

```
{'!', 'T', 'a', 'e', 'i', 'l', 'm', 'n', 'o', 'r', 't'}
```

In summary, while Python doesn't have tuple comprehensions directly, we can use generator expressions to achieve similar functionality, which can then be converted into tuples, lists, or sets as needed. Generator expressions offer both memory efficiency and flexibility by generating values on the fly rather than storing them in memory.

As we move forward, we'll encounter more examples of how generators can be powerful tools for optimizing performance, especially when dealing with large datasets. For now, this introduction should give you a solid foundation for using generator expressions in simple cases.