Control flow, list comprehensions, and functions

Lecture 03

Dr. Colin Rundel

Control Flow

Conditionals

Python supports typical if / else style conditional expressions,

```
1 x = 42
2
3 if x < 0:
4     print("X is negative")
5 elif x > 0:
6     print("X is positive")
7 else:
8     print("X is zero")
```

```
X is positive
```

```
1 x = 0
2
3 if x < 0:
4     print("X is negative")
5 elif x > 0:
6     print("X is positive")
7 else:
8     print("X is zero")
```

```
X is zero
```

Significant whitespace

This is a fairly unique feature of Python - expressions are grouped together via *indenting*. This is relevant for control flow (`if`, `for`, `while`, etc.) as well as function and class definitions and many other aspects of the language.

Indents should be 2 or more spaces (4 is generally preferred based on PEP 8) or tab character(s) - generally your IDE will handle this for you.

If there are not multiple expressions then indenting is optional, e.g.

```
1 if x == 0: print("X is zero")
```

```
X is zero
```

Conditional scope

Conditional expressions do not have their own scope, so variables defined within will be accessible / modified outside of the conditional.

This is also true for other control flow constructs (e.g. `for`, `while`, etc.)

```
1 s = 0
2 s
```

```
0
```

```
1 if True:
2     s = 3
3
4 s
```

```
3
```

`while` loops

will repeat until the provided condition evaluates to `False`,

```
i = 17
seq = [i]

while i != 1:
    if i % 2 == 0:
        i /= 2
    else:
        i = 3*i + 1

    seq.append(i)

seq
```

```
[17, 52, 26.0, 13.0, 40.0, 20.0, 10.0, 5.0, 16.0, 8.0, 4.0, 2.0, 1.0]
```

`for` loops

iterate over the elements of a *sequence*,

```
1 for w in ["Hello", "world!"]:
2     print(w, ":", len(w))
```

```
Hello : 5
world! : 6
```

```
1 sum = 0
2 for v in (1,2,3,4):
3     sum += v
4 sum
```

```
10
```

```
1 res = []
2 for c in "abc123def567":
3     if (c.isnumeric()):
4         res.append(int(c))
5 res
```

```
[1, 2, 3, 5, 6, 7]
```

```
1 res = []
2 for i in range(0,10):
3     res += [i]
4 res
```

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

`break` and `continue`

allow for either an early loop exit or a step to the next iteration respectively,

```
1 for i in range(1,10):
2     if i % 3 == 0:
3         continue
4
5     print(i, end=" ")
```

```
1 2 4 5 7 8
```

```
1 for i in range(1,10):
2     if i % 3 == 0:
3         break
4
5     print(i, end=" ")
```

```
1 2
```

loops and `else`

Both `for` and `while` loops can also include an `else` clauses which execute when the loop is completes by either fully iterating (`for`) or meetings the `while` condition, i.e. when `break` is not used.

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break
    else:
        print(n, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

pass

is a placeholder expression that does nothing, it can be used when an expression is needed syntactically.

```
x = -3

if x < 0:
    pass
elif x % 2 == 0:
    print("x is even")
elif x % 2 == 1:
    print("x is odd")
```

List comprehensions

Basics

List comprehensions provides a concise syntax for generating lists (or other sequences) via iteration over another list (or sequences).

```
res = []
for x in range(10):
    res.append(x**2)
res
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
[x**2 for x in range(10)]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Since it uses the for loop syntax, any sequence / iterable object is fair game:

```
[x**2 for x in [1,2,3]]
```

```
[1, 4, 9]
```

```
[x**2 for x in (1,2,3)]
```

```
[1, 4, 9]
```

```
[c.lower() for c in "Hello World!"]
```

```
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '!']
```

Using `if`

List comprehensions can include a conditional clause(s) to filter the input list / object,

```
1 [x**2 for x in range(10) if x % 2 == 0]
```

```
[0, 4, 16, 36, 64]
```

```
1 [x**2 for x in range(10) if x % 2 == 1]
```

```
[1, 9, 25, 49, 81]
```

The comprehension can include multiple `if` statements (which are combined via `and`)

```
1 [x**2 for x in range(10) if x % 2 == 0 if x % 3 ==0]
```

```
[0, 36]
```

```
1 [x**2 for x in range(10) if x % 2 == 0 and x % 3 ==0]
```

```
[0, 36]
```

Multiple `for` keywords

Similarly, the comprehension can also contain multiple `for` statements which is the equivalent of nested `for` loops,

```
res = []
for x in range(3):
    for y in range(3):
        res.append((x,y))
res
```

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

```
[(x, y) for x in range(3) for y in range(3)]
```

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

zip

Is a useful function for “joining” the elements of multiple sequences (so they can be jointly iterated over),

```
1 x = [1,2,3]
2 y = [3,2,1]
3 z = zip(x, y)
4 z
```

```
<zip object at 0x12a1d3a80>
```

```
1 list(z)
```

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

```
1 [a**b for a,b in zip(x,y)]
```

```
[1, 4, 3]
```

```
1 [b**a for a,b in zip(x,y)]
```

```
[3, 4, 1]
```

The zip object is an example of a generator which we will talk more about later on.

`zip` and length mismatches

The length of the shortest sequence will be used, additional elements will be ignored (silently)

```
1 x = [1,2,3,4]
2 y = range(3)
3 z = "ABCDE"
4
5 list(zip(x,y))
```

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

```
1 list(zip(x,z))
```

```
[(1, 'A'), (2, 'B'), (3, 'C'), (4, 'D')]
```

```
1 list(zip(x,y,z))
```

```
[(1, 0, 'A'), (2, 1, 'B'), (3, 2, 'C')]
```

Exercise 1

Using list comprehensions, complete the following tasks:

- Create a list containing tuples of x and y coordinates of all points of a regular grid for $x \in [0, 10]$ and $y \in [0, 10]$.
- Count the number of points where $y > x$.
- Count the number of points x or y is prime.

Functions

Basic functions

Functions are defined using `def`, arguments can be defined with out without default values.

```
1 def f(x, y=2, z=3):
2     print(f"x={x}, y={y}, z={z}")
```

```
1 f(1)
```

```
x=1, y=2, z=3
```

```
1 f(1,z=-1)
```

```
x=1, y=2, z=-1
```

```
1 f("abc", y=True)
```

```
x=abc, y=True, z=3
```

```
1 f(z=-1, x=0)
```

```
x=0, y=2, z=-1
```

```
1 f()
```

```
TypeError: f() missing 1 required
positional argument: 'x'
```

Arguments with defaults *must* must follow the argument(s) without defaults

`return` statements

Functions must explicitly include a `return` statement to return a value.

```
1 def f(x):
2     x**2
3
4 f(2)
5 type(f(2))
```

```
<class 'NoneType'>
```

```
1 def g(x):
2     return x**2
3
4 g(2)
```

```
4
```

```
1 type(g(2))
```

```
<class 'int'>
```

Functions can contain multiple `return` statements

```
1 def is_odd(x):
2     if x % 2 == 0: return False
3     else:          return True
4
5 is_odd(2)
```

```
False
```

```
1 is_odd(3)
```

```
True
```

Multiple return values

Functions can return multiple values using a tuple or list,

```
1 def f():
2     return (1,2,3)
3 f()
```

```
(1, 2, 3)
```

```
1 def g():
2     return [1,2,3]
3 g()
```

```
[1, 2, 3]
```

If multiple values are present and not in a sequence, then it will default to a tuple,

```
1 def h():
2     return 1,2,3
3
4 h()
```

```
(1, 2, 3)
```

```
1 def i():
2     return 1, [2, 3]
3
4 i()
```

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

Docstrings

A common practice in Python is to document functions (and other objects) using a doc string - this is a short concise summary of the objects purpose. Docstrings are specified by supplying a string as the very line in the function definition.

```
1 def f():
2     "Hello, I am the function f() \
3 and I don't do anything"
4 
5     pass
6 
7 f.__doc__
```

```
"Hello, I am the function f() and I don't do
anything"
```

```
1 def g():
2     """This function also
3 does absolutely nothing.
4 """
5 
6     pass
7 
8 g.__doc__
```

```
'This function also \ndoes absolutely nothing.\n'
```

Using docstrings

```
1 print(max.__doc__)
```

```
max(iterable, *[, default=obj, key=func]) -> value
max(arg1, arg2, *args, *[, key=func]) -> value

With a single iterable argument, return its biggest item. The
default keyword-only argument specifies an object to return if
the provided iterable is empty.
With two or more arguments, return the largest argument.
```

```
1 print(str.__doc__)
```

```
str(object='') -> str
str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or
errors is specified, then the object must expose a data buffer
that will be decoded using the given encoding and error handler.
Otherwise, returns the result of object.__str__() (if defined)
or repr(object).
encoding defaults to sys.getdefaultencoding().
errors defaults to 'strict'.
```

```
1 print("".lower.__doc__)
```

```
Return a copy of the string converted to lowercase.
```

Argument order

In Python the argument order matters - positional arguments must always come before keyword arguments.

```
1 def f(x, y, z):
2     print(f"x={x}, y={y}, z={z}")
```

```
1 f(1,2,3)
```

```
x=1, y=2, z=3
```

```
1 f(x=1,y=2,z=3)
```

```
x=1, y=2, z=3
```

```
1 f(1,y=2,z=3)
```

```
x=1, y=2, z=3
```

```
1 f(y=2,x=1,z=3)
```

```
x=1, y=2, z=3
```

```
1 f(x=1,y=2,3)
```

```
positional argument follows keyword argument
(<string>, line 1)
```

```
1 f(x=1,2,z=3)
```

```
positional argument follows keyword argument
(<string>, line 1)
```

```
1 f(1,2,z=3)
```

```
x=1, y=2, z=3
```

Positional vs keyword arguments

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                   |
        |        Positional or keyword    |
        |                                 - Keyword only
         -- Positional only
```

For the following function `x` can only be passed by position and `z` only by name

```
1 def f(x, /, y, *, z):
2     print(f"x={x}, y={y}, z={z}")
```

```
1 f(1,1,z=1)
```

```
x=1, y=1, z=1
```

```
1 f(1,y=1,z=1)
```

```
x=1, y=1, z=1
```

```
1 f(1,1,1)
```

```
TypeError: f() takes 2 positional arguments but 3 were given
```

```
1 f(x=1,y=1,z=1)
```

```
TypeError: f() got some positional-only arguments passed as keyword arguments: 'x'
```

Variadic arguments

If the number of arguments is unknown / variable it is possible to define variadic functions using `*` or `**`. The former is for unnamed arguments which will be treated as a `tuple`, the latter is for named arguments which will be treated as a `dict`.

```
1 def paste(*x, sep=" "):
2     return sep.join(x)
```

```
1 paste("A")
```

```
'A'
```

```
1 paste("A","B","C")
```

```
'A B C'
```

```
1 paste("1","2","3",sep=",")
```

```
'1,2,3'
```

Anonymous functions

are defined using the `lambda` keyword, they are intended to be used for very short functions (syntactically limited to a single expression, and do not need a return statement)

```
1 def f(x,y):
2     return x**2 + y**2
3
4 f(2,3)
```

```
13
```

```
1 type(f)
```

```
<class 'function'>
```

```
1 g = lambda x, y: x**2 + y**2
2
3
4 g(2,3)
```

```
13
```

```
1 type(g)
```

```
<class 'function'>
```

Function annotations (type hinting)

Python nows supports syntax for providing metadata around the expected type of arguments and the return value of a function.

```
1 def f(x: str, y: str, z: str) -> str:
2     return x + y + z
```

These annotations are stored in the `__annotations__` attribute

```
1 f.__annotations__
```

```
{'x': <class 'str'>, 'y': <class 'str'>, 'z': <class 'str'>, 'return': <class
'str'>}
```

But doesn't actually do anything at runtime:

```
1 f("A","B","C")
```

```
'ABC'
```

```
1 f(1,2,3)
```

6

Exercise 2

1. Write a function, `kg_to_lb`, that converts a list of weights in kilograms to a list of weights in pounds (there a 1 kg = 2.20462 lbs). Include a doc string and function annotations.

2. Write a second function, `total_lb`, that calculates the total weight in pounds of an order, the input arguments should be a list of item weights in kilograms and a list of the number of each item ordered.