

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 s
```

3

while loops

repeats until the condition expression evaluates to `False`,

```
1 i = 17
2 seq = [i]
3
4 while i != 1:
5     if i % 2 == 0:
6         i /= 2
7     else:
8         i = 3*i + 1
9
10    seq.append(i)
11
12 seq
```

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

for loops

iterates 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 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` clause which executes when the loop completes by either fully iterating (`for`) or meeting the `while` condition, i.e. when `break` is not used.

```
1 for n in range(2, 10):
2     for x in range(2, n):
3         if n % x == 0:
4             print(n, 'equals', x, '*', n//x)
5             break
6     else:
7         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.

```
1 x = -3
2 if x < 0:
3     pass
4 elif x % 2 == 0:
5     print("x is even")
6 elif x % 2 == 1:
7     print("x is odd")
```

List comprehensions

Basics

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

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

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

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

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

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

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

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

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

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

```
1 [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 / sequence,

```
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 (equivalent to using `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 are equivalent to nested `for` loops,

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

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

```
1 [(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),`

zip

`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 0x119f74580>
```

```
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]
```

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"
```

```
1 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 where x or y is prime.

Functions

Basic functions

Functions are defined using `def`, arguments can optionally have default values. (Arguments with defaults *must* must follow the arguments without defaults)

```
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 argument

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 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
```

```
1 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 h()
```

(1, 2, 3)

```
1 def i():  
2     return 1, [2, 3]  
3 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 object's purpose. Docstrings are specified by supplying a string as the first 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
```

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

```
'This function also \ndoes absolutely noth
```

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 argume

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

positional argument follows keyword argume

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

x=1, y=2, z=3

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

Anonymous functions are defined using the `lambda` keyword. They are intended 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 now 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 (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.

Classes

Basic syntax

These are the basic components of Python's object oriented system - we've been using them regularly all over the place and will now look at how they are defined and used.

```
1 class rect:
2     """An object representing a rectangle"""
3
4     # Attributes
5     p1 = (0,0)
6     p2 = (1,2)
7
8     # Methods
9     def area(self):
10        return ((self.p1[0] - self.p2[0]) *
11                (self.p1[1] - self.p2[1]))
12
13    def set_p1(self, p1):
14        self.p1 = p1
15
16    def set_p2(self, p2):
17        self.p2 = p2
```

```
1 x = rect()
```

```
1 x.area()
```

2

```
1 x.set_p2((1,1))
2 x.area()
```

1

```
1 x.p1
```

(0, 0)

```
1 x.p2
```

(1, 1)

```
1 x.p2 = (0,0)
2 x.area()
```

0

Instantiation (constructors)

When instantiating a class object (e.g. `rect()`) we invoke the `__init__()` method if it is present in the classes' definition.

```
1 class rect:
2     """An object representing a rectangle"""
3
4     # Constructor
5     def __init__(self, p1 = (0,0), p2 = (1,1))
6         self.p1 = p1
7         self.p2 = p2
8
9     # Methods
10    def area(self):
11        return ((self.p1[0] - self.p2[0]) *
12                (self.p1[1] - self.p2[1]))
13
14    def set_p1(self, p1):
15        self.p1 = p1
16
17    def set_p2(self, p2):
18        self.p2 = p2
```

```
1 x = rect()
2 x.area()
```

1

```
1 y = rect((0,0), (3,3))
2 y.area()
```

9

```
1 z = rect((-1,-1))
2 z.p1
```

(-1, -1)

```
1 z.p2
```

(1, 1)

Method chaining

We've seen a number of objects (e.g., Pandas DataFrames) that allow for method chaining to construct a pipeline of operations. We can achieve the same by having our class methods return itself via `self`.

```
1 class rect:
2     """An object representing a rectangle"""
3
4     # Constructor
5     def __init__(self, p1 = (0,0), p2 = (1,1)):
6         self.p1 = p1
7         self.p2 = p2
8
9     # Methods
10    def area(self):
11        return ((self.p1[0] - self.p2[0]) *
12                (self.p1[1] - self.p2[1]))
13
14    def set_p1(self, p1):
15        self.p1 = p1
16        return self
17
18    def set_p2(self, p2):
19        self.p2 = p2
20        return self
```

```
1 rect().area()
```

1

```
1 rect().set_p1((-1,-1)).area()
```

4

```
1 ( rect()
2   .set_p1((-1,-1))
3   .set_p2((2,2))
4   .area()
5 )
```

9

Class object string formatting

All class objects have a default print method / string conversion method, but the default behavior is not very useful,

```
1 print(rect())
```

```
<__main__.rect object at 0x10da8a450>
```

```
1 str(rect())
```

```
'<__main__.rect object at 0x119f7b920>'
```

Both of the above are handled by the `__str__()` method which is implicitly created for our class - we can override this,

```
1 def rect_str(self):  
2     return f"Rect[{self.p1}, {self.p2}] => area={self.area()}"  
3  
4 rect.__str__ = rect_str
```

```
1 rect()
```

```
<__main__.rect object at 0x11972f7a0>
```

```
1 print(rect())
```

```
Rect[(0, 0), (1, 1)] => area=1
```

```
1 str(rect())
```

```
'Rect[(0, 0), (1, 1)] => area=1'
```

Class representation

There is another special method which is responsible for printing the object (see `rect()` above) called `__repr__()` which is responsible for printing the classes representation. If possible this is meant to be a valid Python expression capable of recreating the object.

```
1 def rect_repr(self):  
2     return f"rect({self.p1}, {self.p2})"  
3  
4 rect.__repr__ = rect_repr
```

```
1 rect()
```

```
rect((0, 0), (1, 1))
```

```
1 repr(rect())
```

```
'rect((0, 0), (1, 1))'
```

Inheritance

Part of the object oriented system is that classes can inherit from other classes, meaning they gain access to all of their parent's attributes and methods. We will not go too in depth on this topic beyond showing the basic functionality.

```
1 class square(rect):  
2     pass
```

```
1 square()
```

```
rect((0, 0), (1, 1))
```

```
1 square().area()
```

```
1
```

```
1 square().set_p1((-1,-1)).area()
```

```
4
```

Overriding methods

```
1 class square(rect):
2     def __init__(self, p1=(0,0), l=1):
3         assert isinstance(l, (float, int)), \
4             "l must be a number"
5
6         p2 = (p1[0]+l, p1[1]+l)
7
8         self.l = l
9         super().__init__(p1, p2)
10
11     def set_p1(self, p1):
12         self.p1 = p1
13         self.p2 = (self.p1[0]+self.l, self.p1[1]+self.l)
14         return self
15
16     def set_p2(self, p2):
17         raise RuntimeError("Squares take l not p2")
18
19     def set_l(self, l):
20         assert isinstance(l, (float, int)), \
21             "l must be a number"
22
23         self.l = l
```

```
1 square()
```

```
square((0, 0), 1)
```

```
1 square().area()
```

```
1
```

```
1 square().set_p1((-1,-1)).area()
```

```
1
```

```
1 square().set_l(2).area()
```

```
4
```

```
1 square((0,0), (1,1))
```

```
AssertionError: l must be a number
```

```
1 square().set_l((0,0))
```

```
AssertionError: l must be a number
```

```
1 square().set_p2((0,0))
```

```
RuntimeError: Squares take l not p2
```

Class attributes

We can examine all of a classes' methods and attributes using `dir()`,

```
1 [
2   dir(rect)
3 ]
```

```
[['_class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__'
```

Where did `p1` and `p2` go?

```
1 [  
2   dir(rect())  
3 ]
```

```
[['_class_', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__']
```