Control Flow

Conditionals

Python supports tradition if / else style conditional expressions,

```
x = 42

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

```
## X is positive
```

```
x = 0

if x < 0:
    print("X is negative")
elif x > 0:
    print("X is positive")
else:
    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.

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

If there are not multple expression then indenting is optional, e.g.

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

```
s = 0
if True:
    s = 3

s
```

```
## 3
```

While loops

Repeat until the given 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]
```

Anyone recognize what this is an example of?

For loops

Iterates over the elements of a sequence

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

```
## Hello 5
## world! 6
```

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

```
## 10
```

```
res = []
for c in "abc123def567":
    if (c.isnumeric()):
        res.append(int(c))
res
```

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

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

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

break **and** continue

Allow early loop exit or step to next iteration respectively,

```
for i in range(10):
    if i == 5:
        break
    print(i, end=" ")
```

```
## 0 1 2 3 4
```

```
print()
```

```
for i in range(10):
    if i % 2 == 0:
        continue

    print(i, end=" ")
```

```
## 1 3 5 7 9
```

```
print()
```

loops and `else`?

Both `for` and `while` loops can also have `else` clauses which execute when the loop is terminated by fully iterating (`for`) or meetings the `while` condition, i.e. when `break` does not execute.

```
# From python tutorial - Section 4.4
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
```

List comprehensions

Basics

List comprehensions provides a concise syntax for generating lists

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

```
[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 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,

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

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

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

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

The comprehension can include multiple `if` statements,

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

```
## [0, 36]
```

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

```
## [0, 36]
```

Multiple `for`s

Similarly, the comprehension can also contain multiple `for` statements,

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

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

```
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,
```

zip

This is a useful function for "joining" elements of a sequence,

```
x = [1,2,3]
y = [3,2,1]
```

```
z = zip(x, y)
z
```

```
## <zip object at 0x10ef0e400>
```

```
list(z)
```

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

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

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

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

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

`zip` and length mismatches

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

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

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

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

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

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

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

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

Functions

Basic functions

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

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

```
f(1)
```

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

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

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

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

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

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

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

```
f()
```

```
## Error in py_call_impl(callable, dots$args, dots$keyw
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

`return` statements

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

```
def f(x):
    x**2
```

```
f(2)
```

```
type(f(2))
```

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

```
def g(x):
    return x**2
```

```
g(2)
```

```
## 4
```

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

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

Functions can contain multiple `return` statements

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

is_odd(2)
```

```
## False
```

Multiple return values

Functions can return multiple values using a tuple or list,

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

f()
```

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

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

g()
```

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

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

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

h()
```

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

Doc strings

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

```
def f():
    "Hello."

    pass

f.__doc__
```

```
## 'Hello.'
```

```
def g():
    """This function does
    absolutely nothing.
    """

    pass

g.__doc__
```

```
## 'This function does\n    absolutely nothing.\n    '
```

Variadic arguments

If the number of arguments is unknown it is possible to define to define variadic functions

```
def paste(*args, sep=" "):
    return sep.join(args)
```

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

```
## 'A'
```

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

```
## 'A B C'
```

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

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

Positional and/or 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

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

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

```
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: f() takes 2 positional arguments but 3 were given
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

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

```
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: f() got some positional-only arguments passed as keyword arguments: 'x
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

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

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

Based on Python tutorial Sec 4.8.3

Anonymous functions

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

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

f(2,3)
```

```
## 13
```

```
type(f)
```

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

```
g = lambda x, y: x**2 + y**2


g(2,3)
```

```
## 13
```

```
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.

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

These annotations are stored in the `__annotations__` attribute

```
f.__annotations__
```

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

But doesn't actually do anything at runtime:

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

```
## 'ABC'
```

```
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.