# F28LL Programming Languages (GA)

Joe Wells

2021-09-13

# Contents

# Course overview

- Course descriptor: https://www.hw.ac.uk/documents/pams/202122/F28LL_202122.pdf
- In a change this year, we will present programming language concepts mainly via the Python programming language. The full details are not yet finalized.
- The coursework due dates for semester 1 will be finalized shortly.

# Resources

- www.python.org (main Python website)
- Python Setup and Usage
- The Python Tutorial
- Glossary
- Python Frequently Asked Questions (programming; design and history; library and extension; Python on Windows)
- The Python Standard Library (a treasure trove)
- The Python Language Reference
- The Python Wiki (IDEs; editors)
- the “Users” category at “Discussions on Python.org” (a Discourse forum)
- Python Forum

# Python

- Python is named after *Monty Python's Flying Circus*
- Open source
- Highly portable
- First version was made available 1990
- Current stable version is 3.9

# Python 3 vs Python 2

- Python 3 has been stable and fully supported since Python 3.0's final release in 2008.
- Python 2 support ceased at the end of 2019 and even security holes are no longer fixed.
- Python 3 has many new concepts and drops some old ones. See `docs.python.org/3.0/whatsnew/3.0.html` for details.
- Warning: Python 2 samples online might not run on Python 3.
- The `2to3` tool can help convert Python 2 code to Python 3.
- Notable issues are:
  - In Python 3, `print` is just a normal function. Parentheses must be used, i.e., `print(x)` NOT `print x`.
  - Focus on *iterators*: functions that generate sequences (e.g., `map`) now return iterators that can potentially delay producing each item in the sequence until the item is requested.

# Runtime behaviour

- Python source code is compiled by CPython to bytecode — a portable machine code designed for execution by a virtual machine, also called an interpreter. As of CPython 3.9, bytecode is actually 2 bytes per instruction, so it is actually “2-byte-code”.
- Compilation is performed transparently: your programs ‘just run’. CPython either compiles Python to bytecode, or finds a cache of previously compiling the same code, and executes the bytecode.
- CPython manages memory automatically by reference counting and detection of unreachable objects (generational garbage collection). This is somewhat like Java, and unlike C, where memory is usually explicitly allocated and deallocated.
- CPython programs rarely crash due to pointer errors or hardware rule violations *(e.g., “segmentation fault”)*. Crashes are generally orderly and well defined results of raised exceptions.

# Runtime behaviour

You are expected to know, in general terms, about bytecode and automatic memory management.
E.g.:

- *"Q. Explain why C programs often suffer from memory leaks and Python programs generally do not."*
- *"Q. Python and Java compile to bytecode. Explain in general terms what the previous sentence means."*

We're looking for general knowledge (not specific expertise). Read up on this.

# Language features

- Classes with multiple inheritance.
- Everything is an object in some class (very object-oriented).
- Higher-order functions (like Haskell, OCaml, SML, Scheme).
- Dynamic typing.
- Exceptions (like Java).
- Static scoping.
- Modules and packages.
- Overloading of almost all operators and many built-in functions.
- Structured programming (e.g., `while`, `if`..`else`, `for`..`in`).
- Clause structure given by indentation (like Haskell).

# Data-types

- Number classes: `bool`, `int`, `float`, `complex`
- Sequence classes: `list`, `tuple`, `range`
  - String-like classes: `str`, `bytes`
- Set-like classes: `set`, `frozenset`
- Mapping classes: `dict`
- C extensions can define new data-types.
- Python code can model arbitrary data-structures using classes.

# Data-types

- You are expected to be familiar with the `bool`, `int`, `float`, `str`, `bytes`, `list`, `tuple`, `range`, `set`, `frozenset`, and `dict` types.
- You are expected to be able to explain what they are and how they differ.

E.g.:

- Q: Explain the difference between `set` and `frozenset`, with specific reference to the `x.issubset(y)` and `x.add(i)` methods (3 marks).
- A: `set` is mutable, `frozenset` is immutable (1 mark). Thus if `x` has `set` type then it supports both methods (1 mark), whereas if `x` has type `frozenset` then it supports `issubset` but not `add` (1 mark).

See Slide 88 and surrounding slides.

# Why Python?

- Code 2 to 10 times shorter than C#, C++, Java.
- Code is easy to comprehend.
- Good for web scripting.
- Scientific applications: numerical computation (NumPy), AI (TensorFlow), natural language processing, data visualisation (Pandas), etc.
- Used in the Raspberry Pi.
- Rich libraries for XML, databases, graphics, etc.
- Web content management (Zope/Plone).
- Email distribution lists (GNU Mailman).
- PyPy is a faster implementation of Python 3.7.
- IronPython is a Python compiler targeting the .Net platform.

# Why Python?

- Active community.
- Good libraries.
- Widely used in teaching institutes.
- Supports many aspects of functional programming.

# The Python interactive shell

- The UNIX (e.g., Linux, MacOS) terminal command to start Python might be `python3`, `/usr/bin/python3`, or `python3.9`.
- The Windows command might be `python` or `py`.
- With no arguments, this starts an interactive Python shell:

```
$ python3
```

```
Python 3.5.3 (default, Apr  5 2021, 09:00:41)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for mor
>>> print(sum([1, 2, 3, 4]))
10
```

- To exit, enter "`quit()`" or tell the terminal "end of file":
  - On UNIX: type Control-D at the beginning of a line.
  - On Windows: type Control-Z Enter.

# The entire program on the command line

- You can execute a short 1-line Python script like this on UNIX:

```
$ python3 -c 'print(sum([1, 2, 3, 4]))'
10
```

- Using a typical UNIX shell (e.g., bash), the apostrophe maximally protects the Python program. Use a quote character instead to allow shell expansions on the argument.
- On Windows you need to use the quote character instead of the apostrophe:

```
> python -c "print(sum([1, 2, 3, 4]))"
10
```

# Putting a program in a file

- Suppose this is in the file `xyzzy.py`:

```
#!/usr/bin/env python3
print(sum([1, 2, 3, 4]))
```

- If the file is in the current directory, this runs it (and prints "`10`"):

```
$ python3 xyzzy.py
```

- On UNIX, you can make the file executable like this:

```
$ chmod a+x xyzzy.py
```

- On UNIX, if the file is executable and in the current directory, this runs it:

```
$ ./xyzzy.py
```

- On UNIX, if the file is executable and in a directory listed in your `PATH` environment variable, this runs it:

```
$ xyzzy.py
```

# Launching Python

Python is often started from a terminal/console window command line:

- You can start an interactive Python shell like this:
  `$ python3`
  - To exit the interactive Python shell, enter "`quit()`" or tell the terminal "end of file" (UNIX: Ctrl-D; Windows: Ctrl-Z Enter).
- You can compile and execute a Python program file like this:
  `$ python3 xyzzy.py`
  - Or just enter the command "`xyzzy.py`". (On UNIX the file must be executable.) The first line of the file should look like this:

    ```
    #!/usr/bin/env python3
    ```
- You can execute a short 1-line Python script like this:

```
$ python3 -c 'print(sum([1, 2, 3, 4]))'
10
```

# Python shell interaction

```
$ python3
>>>                          # A comment
>>> x = "Hello World"        # An assignment
>>> x                        # What's the value of x?
'Hello World'
>>> print(x)
Hello World
>>> type(x)                  # What's the type of x?
<class 'str'>
>>> x * 3                    # Concatenate x with x and x
'Hello WorldHello WorldHello World'
```

# Numbers

```
>>> 2 + 2
4
>>> 7 / 3               # / never returns int
2.3333333333333335
>>> 7 / -3
-2.3333333333333335
>>> 7 // 3              # integer division (floor variant)
2
>>> 7 // -3
-3
>>> 10**8               # exponentiation: “to the power of”
100000000
```

# More fun with numbers

```
>>> x = 7 / 3; x
2.3333333333333335
>>> x.is_integer()
False
>>> x.__round__
<built-in method __round__ of float object at 0x767e4e830a98
>>> x.__round__()
2
>>> type(x)
<class 'float'>
>>> type(x.__round__())
<class 'int'>
>>> x.as_integer_ratio()  # not (7,3), floats imprecise:
(5254199565265579, 2251799813685248)
```

Handy tip: Type ‘x.’ then double-tab to list methods of x.

# Numeric precision: `int` v.s `float`

`int`s have unlimited precision.

`float`s are C `double`s (e.g., 52-bit mantissa, 11-bit exponent, sign bit).

```
>>> 10**30          # integer computation
1000000000000000000000000000000
>>> int(1E30)       # float representation; convert to integer
1000000000000000019884624838656
>>> int(1E30) - 10**30  # pretty big rounding error
19884624838656
```

$10^{30}$ is in amongst 140 trillion `int`s in a row without matching `float`s:

```
>>> m=7105427357601001; e=47; x=m*2**e; y=(m+1)*2**e
>>> x                # the nearest float less than 10**30
999999999999997879147136483328
>>> y                # the nearest float greater than 10**30
1000000000000000019884624838656
>>> y - x            # huge 2**47 gap between neighbor floats
140737488355328
```

# More fun with numbers

More innocent fun:

```
>>> def toweri(n):
...     if n == 0:
...         return 2
...     else:
...         return 2**(toweri(n - 1))  # recursive!
...
>>> toweri(3)       # What does it calculate?
65536
>>> # toweri(4)     # Answer several pages long (try it)
>>> # toweri(5)     # Crashes my machine!
```

Python eats memory, trying to calculate `toweri(5)` with full precision. Let's try it with floats instead . . .

# More fun with numbers

```
>>> def towerf(n):
...     if n == 0:
...         return 2.0
...     else:
...         return 2.0**(towerf(n - 1))
...
>>> towerf(3)
65536.0
>>> towerf(4)
OverflowError: (34, 'Numerical result out of range')
```

In other words, `float` just throws up its hands and gives up.
You are required to know that `int`egers are infinite precision in Python and `float`s aren't, and e.g., predict the behaviour of the two `tower` functions above.

# Arbitrary precision rational numbers

For the full rational numbers, Python's standard library has the `fractions` module. For ludicrously high precision floating-point decimal numbers, Python's standard library has the `decimal` module. For unlimited precision floating-point binary numbers, there is the `mpmath` package.

```
$ sudo apt-get install python3-mpmath
$ python3 -q
>>> from mpmath import *
>>> mp.dps = 50; mp.pretty = True
>>> fdiv(7, 3)
2.3333333333333333333333333333333333333333333333
>>> +pi
3.1415926535897932384626433832795028841971693993751
>>> +sqrt(2)
1.4142135623730950488016887242096980785696718753769
```

That's more like it!

# Assignment

- Most variables don't need to be declared (scripting language):

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

- Assignment of iterables to a *target list*:

```
>>> width, height = height, width + height
```

- Assigning the same object to multiple targets:

```
>>> x = y = z = 0    # Zero out x, y, and z
>>> x
0
>>> z
0
>>> x = 0 = y = z    # Worth a try
  File "<stdin>", line 1
SyntaxError: can't assign to literal
```

# Back to floating point

- Many arithmetic operations yield `float` results even with one `int` input:

```
>>> 1 / 1        # division with / never returns int
1.0
>>> 3 * 4.0
12.0
```

- Exponent notation: `1e0`, `1.0e+1`, `1e-1`, `.1e-2`
- Same precision and range as `double` in C.

```
>>> 1e-323
1e-323
>>> 1e-324       # too small for float
0.0
```

# Back to floating point

```
>>> 1e-323
1e-323
>>> 1e-324      # too small for float
0.0
```

If we loaded `mpmath` we can use its `power` function:

```
>>> from mpmath import power
>>> power(10, -324)
1.0e-324
>>> power(10, -325)
1.0e-325
```

# Further arithmetic operations

- Remainder:

```
>>> 4 % 3
1
>>> -4 % 3
2
>>> 4 % -3
-2
>>> -4 % -3
-1
>>> 3.9 % 1.3
1.2999999999999998
```

- Division and Floor:

```
>>> 7.0 // 4.4
1.0
```

# complex numbers

- Imaginary numbers have the suffix `j`.

```
>>> 1j * complex(0, 1)
(-1+0j)
>>> complex(-1, 0) ** 0.5
(6.123233995736766e-17+1j)
```

- Real and imaginary components:

```
>>> a = 1.5 + 0.5j
>>> a.real + a.imag
2.0
```

- Absolute value is also defined on `complex`.

```
>>> abs(3 + 4j)
5.0
```

You're not expected to know complex numbers by heart. In an exam you might get relevant definitions and a question using them.

# Bit operations

- *Left-* (<<) and *right-shift* (>>)

```
>>> 1 << 16
65536
```

- Bitwise *and* (&), *or* (|), *xor* (^) and *negation* (~).

```
>>> 254 & 127
126
>>> 254 | 127
255
>>> 254 ^ 127
129
>>> ~0
-1
```

# Bit operations

Binary representation using `bin`, go back using `int`.

```
>>> bin(1 << 16)
'0b10000000000000000'
>>> type(bin(1 << 16))
<class 'str'>
>>> int(bin(1 << 16), 2)
65536
>>> bin(0b11111110 & 0b01111111) # loses leading zero
'0b1111110'
>>> bin(0b11111110 | 0b01111111)
'0b11111111'
>>> bin(0b11111110 ^ 0b01111111)
'0b10000001'
```

How to keep leading zeroes? Use `format` instead:

```
>>> format(0b11111110 & 0b01111111, '#010b')
'0b01111110'
```

# strings

- Type: `str`.

```
>>> type("Hello world")
<class 'str'>
```

- Single- and double-quotes can be used

```
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

# Escape sequences in string literals

```
\\          backslash
\'          single quote
\"          double quote
\t          tab
\n          newline
\r          carriage return
\b          backspace
```

# Multi-line string literals

```
>>> print ("This is a rather long string containing\n\
... several lines of text as you would do in C.\n\
...     Whitespace at the beginning of the line is\
...  significant.")
This is a rather long string containing
several lines of text as you would do in C.
    Whitespace at the beginning of the line is significant.
```

# Triple-quote string literals

Multi-line string including line-breaks:

```
>>> print ("""
... Usage: thingy [OPTIONS]
...      -h                     Display this usage message
...      -H hostname            Hostname to connect to
... """)

Usage: thingy [OPTIONS]
     -h                     Display this usage message
     -H hostname            Hostname to connect to
```

# Raw string literals

- An `r` prefix preserves all escape-sequences, although a backslash continues to prevent an immediately following quote character from ending the string literal.

```
>>> print("Hello! \n\"How are you?\"")
Hello!
"How are you?"
>>> print(r"Hello! \n\"How are you?\"")
Hello! \n\"How are you?\"
```

- Raw string literals also have type `str`.

```
>>> type("\n")
<class 'str'>
>>> type(r"\n")
<class 'str'>
```

# Unicode

- Python strings are Unicode:

```
>>> print("a\u0020b")
a b
>>> 'Γει´α σας!'  # “Hello!” in Greek
'Γει´α σας!'
>>> type(_)       # interactive expressions assign to _
<class 'str'>
```

- Python source code must either use the UTF-8 encoding or must have a coding declaration magic comment (and the encoding used must be ASCII-compatible enough for the coding declaration to be read as ASCII).

# string operations

```
"hello"+"world"    "helloworld"         # concatenation
"hello"*3          "hellohellohello"    # repetition
"hello"[0]         "h"                  # indexing
"hello"[-1]        "o"                  # (from end)
"hello"[1:4]       "ell"                # slicing
len("hello")       5                    # size
"hello" < "jello"  True                 # ordering
"e" in "hello"     True                 # substring search
```

# string operations

You are required to understand that string comparison is lexicographic on the underlying Unicode code-point sequences—and that this can have counter-intuitive consequences:

```
>>> "z" < "a"
False
>>> "Z" < "a"
True
>>> "Z" < "aardvark"
True
>>> "z" < "aardvark"
False
>>> "a" < "B"
False
>>> "9" < "10"
False
```

See this page for more on ‘human’ sorting.

# `list` objects (really arrays)

- Lists are *arrays* and they are *mutable*.

```
>>> a = [99, "bottles of beer", ["on", "the", "wall"]]
```

- Sequence operations that work on strings also work on lists.

```
>>> [7] + [8], [5] * 2, a[0], len(a)
([7, 8], [5, 5], 99, 3)
>>> a[-1]
['on', 'the', 'wall']
>>> a[1 :]
['bottles of beer', ['on', 'the', 'wall']]
```

- Elements and segments can be modified.

```
>>> a[0] = 98; a[1 : 2] = ["bottles", "of", "beer"]
>>> a
[98, 'bottles', 'of', 'beer', ['on', 'the', 'wall']]
>>> del a[-1]
>>> a
[98, 'bottles', 'of', 'beer']
```

# More `list` operations

```
>>> a = list(range(5)); a
[0, 1, 2, 3, 4]
>>> a.append(5); a
[0, 1, 2, 3, 4, 5]
>>> a.pop(), a
(5, [0, 1, 2, 3, 4])
>>> a.insert(0, 42); a
[42, 0, 1, 2, 3, 4]
>>> a.pop(0), a
(42, [0, 1, 2, 3, 4])
>>> a.reverse(); a
[4, 3, 2, 1, 0]
>>> a.sort(); a
[0, 1, 2, 3, 4]
```

You are required to understand these methods modify data *in-place* ...

# More `list` operations

. . . and that if you want sorting/reversing without modifying the original data and you don't object if the sorted/reversed sequence is produced lazily, then use functions such as `reversed` or `sorted`:

```
>>> a = ['0','1']; a.reverse()  # a gets reversed in-place
>>> a
['1', '0']
>>> type(['0','1'].reverse())   # reversed then discarded
<class 'NoneType'>
>>> reversed(['0','1'])         # You probably meant this
<list_reverseiterator object at 0x767e4ba13128>
>>> list(reversed(['0','1']))   # force generation
['1', '0']
>>> sorted(reversed(['0','1'])) # Now sort it, just for fun
['0', '1']
```

# More `list` operations

Really thinking about this is enough to make my head hurt. Watch this:

```
>>> a = [0, 1]          # Make a list
>>> a.reverse()         # Reverse it
>>> a                   # Yup, reversed:
[1, 0]
>>> type(a)             # What's its type? list:
<class 'list'>
>>> a.reverse()         # Reverse it again.
>>> a = reversed(a)     # Now reverseD it ...
>>> type(a)             # What's the result type?
<class 'list_reverseiterator'>
>>> a                   # Not a list, an iterator!
<list_reverseiterator object at 0x767e4ba13128>
>>> list(a)             # Force generation
[1, 0]
```

# lists and strings

If `str_list` is a list of strings and `s` is a string, `s.join(str_list)` concatenates the items in `str_list` separated by `s`.
E.g., `','.join(str_list)` is a comma-separated list inside a string.
You are required to understand the code below, including the reason for the error message:

```
>>> a = [99, "bottles of beer", ["on", "the", "wall"]]
>>> a[1] + a[2]
TypeError: Can't convert 'list' object to str implicitly
>>> a[1:2]+a[2]
['bottles of beer', 'on', 'the', 'wall']
>>> ' '.join(a[1 : 2] + a[2])
'bottles of beer on the wall'
>>> # [exp for iter] below is a list comprehension (later)
>>> ', '.join([str(i) for i in range(10,0,-1)])+', liftoff!'
'10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!'
```

# Truth and `booleans`

- False objects include:
  - **`False`**, `0`, `0.0`, `0j`, and other built-in numeric zeroes.
  - **`None`**.
  - `''`, `[]`, `{}`, `()`, and other built-in empty collections.
  - Objects that have a `__bool__` method returning False or that have no `__bool__` method and have a `__len__` method returning 0.
- All other objects are true (also functions and classes!).
- `bool` objects are numbers.

```
>>> issubclass(bool, int)
True
>>> False == 0
True
>>> True == 1
True
```

# Truth and booleans

- Comparison operators (==, <, >, <=, >=, !=, **is**, **in**) can be chained, so `a < b == c > d` stands for `a < b and b == c and c > d`.
- Operators **and** and **or** only evaluate their 2nd argument if needed.

```
>>> print('1st') and print('2nd')
1st
```

- **and** and **or** return one of their arguments (like LISP):

```
>>> '' or 'you' or 'me'
'you'
```

# `while` statements

- Print all Fibonacci numbers up to 100:

```
>>> a, b = 0, 1
>>> while b <= 100:
...     print(b, end=' ')
...     a, b = b, a + b
...
1 1 2 3 5 8 13 21 34 55 89
```

- **IMPORTANT:** Indentation carries semantics in Python:
  - Indentation begins a *suite* (other languages call this a “block”).
  - De-indentation ends a suite.
- A suite of only *simple statements* can go on the same line after a *clause header*:

```
>>> a, b = 0, 1
>>> while b <= 100: print(b, end=' '); a, b = b, a + b
...
1 1 2 3 5 8 13 21 34 55 89
```

# `if` statements

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 5
>>> if x < 0:
...         x = -1
...         print('Sign is Minus')
... elif x == 0:
...         print('Sign is Zero')
... elif x > 0:
...         print('Sign is Plus')
... else:
...         print('Should never see that')
...
Sign is Plus
```

- To reduce indentation, use `elif` instead of `else:` followed by `if`.

# `for` statements

`for` iterates over any *iterable* (e.g., `list`, `tuple`, `range`, `str`, `bytes`, `set`, `frozenset`, `dict`, io.TextIOWrapper).

```
>>> a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print(x, len(x), repr(x))
...
cat 3 'cat'
window 6 'window'
defenestrate 12 'defenestrate'
>>> for x in 'cat':
...     print(x, len(x), repr(x))
...
c 1 'c'
a 1 'a'
t 1 't'
```

# Iterables and iterators

- Each `for` statement loops over an *iterable* object, but what is that?
- x is an *iterable* exactly when `iter(x)` returns an *iterator*.
- y is an *iterator* exactly when `iter(y) == y` and after `next(y)` raises `StopIteration` so does every subsequent `next(y)`.
- For example, every `str` object is an iterable:

```
>>> i = iter('abc'); i
<str_iterator object at 0x767e4b610390>
>>> next(i); next(i); next(i); next(i)
'a'
'b'
'c'
StopIteration
>>> next(i)
StopIteration
```

# Objects of many classes are iterables

- Every `list` object is an iterable yielding the `list`'s members:

```
>>> i = iter([1, 2]); i
<list_iterator object at 0x767e4b5fc828>
>>> next(i); next(i); next(i)
1
2
StopIteration
```

- Every `dict` object is an iterable yielding the `dict`'s keys:

```
>>> i=iter({'supervisor':'Alonzo','student':'Alan'}); i
<dict_keyiterator object at 0x767e4ba16368>
>>> next(i); next(i); next(i)
'student'
'supervisor'
StopIteration
```

# `for` statements

If a `for`-loops's iterable is modified in the loop, anything can happen!

```
>>> a = ['Hi', 'De']
>>> for x in a:
...     print(x, end=' '); a.extend(a[0 : 1])
...     if len(a) > 18: print(); break
...
Hi De Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi Hi
>>> a = ['Hi','De']
>>> for x in a:
...     print(x, end=' '); a.extend(a[0 :])
...     if len(a) > 1000000: print(); break
...
Hi De Hi De Hi De Hi De Hi De Hi De Hi De Hi De Hi De
```

In the second loop, the length of a is doubled at *each cycle*.

# `for` statements

We can create a copy first, e.g., using the slicing `a[:]` or `list`.

```
>>> a = ['Hi', 'De']
>>> for x in a[ : ]: print(x); a.extend(a[0 :])
...                              # Fingers crossed ...
Hi
De
>>> a                            # Why 8 elements?:
['Hi', 'De', 'Hi', 'De', 'Hi', 'De', 'Hi', 'De']
>>> a = ['Hi', 'De']             # Reset a
>>> for x in list(a): print(x); a.extend(a[0 :])
...
Hi
De
>>> a
['Hi', 'De', 'Hi', 'De', 'Hi', 'De', 'Hi', 'De']
```

# range function

range makes a compact, efficient representation of a number sequence.

```
>>> range(10)                # range from 0 to 9
range(0, 10)
>>> type(range(10))          # What's its type?
<class 'range'>
>>> list(range(10))          # expand range to list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(10)[2 : 4]         # slice of the range
range(2, 4)
>>> list(range(10)[2:4]) # slice range then expand to list
[2, 3]
>>> # Move one tiny bracket left:
>>> list(range(10))[2:4] # expand then slice (inefficient)
[2, 3]
```

# range function

```
>>> range(10**7)                # ... so take some big range
range(0, 10000000)
>>> list(range(10**7))[2:4] # Bracket one way, runs slow:
[2, 3]
>>> list(range(10**7)[2:4]) # Bracket the other, runs fast
[2, 3]
```

`list(range(10**10))` froze my system up.

You are expected to understand that `range` does not compute all the individual numbers in the sequence, but rather creates a compact efficient representation of all of those numbers: the two endpoints and the increment between numbers in the sequence.

You are expected to recognise the implications, e.g., for efficiency and limitations on the possible sequences that can be ranges.

# range function

```
>>> list(range(10))           # Can omit start point
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 10))        # Can specify start point
[5, 6, 7, 8, 9]
>>> list(range(0, 10, 3))     # Can specify jump
[0, 3, 6, 9]
>>> list(range(-10, -100, -30))  # Also negative numbers
[-10, -40, -70]
>>> list(range(-10))          # Be careful ...
[]
>>> list(range(0,-10,-1))     # ... you probably meant this!
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

# range function

A typical use is iteration over the indices of an array.

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i], end=' ')
...
0 Mary 1 had 2 a 3 little 4 lamb
```

This can also be done with `enumerate`:

```
>>> list(enumerate(a))
[(0, 'Mary'), (1, 'had'), (2, 'a'), (3, 'little'), (4, 'lamb
>>> for i, s in enumerate(a):
...     print(i, s, end=' ')
...
0 Mary 1 had 2 a 3 little 4 lamb
```

# `for`/`while`-loops: `break`, `continue`, `else`

- **`break`** (as in C), terminates the enclosing loop immediately.
- **`continue`** (as in C), starts the next iteration of the enclosing loop.
- The **`else`**: clause of a **`for`**/**`while`** statement runs exactly when the preceding clauses finish but not due to **`break`** or an exception.

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print (n,'=',x,'*',n//x,sep='',end=' ')
...             break
...     else:       # loop completed, no factor found
...         print (n, 'prime', end=' ')
...
2 prime 3 prime 4=2*2 5 prime 6=2*3 7 prime 8=2*4 9=3*3
```

# The `pass` (null) statement

- The statement `pass` does nothing.
- `pass` is useful when a statement is syntactically required but it doesn't have to do any work.

```
>>> try:
...   for n in range(3, 10):
...     for a in range(10):
...       for b in range(10):
...         for c in range(10):
...           if a**n + b**n == c**n:
...             raise RuntimeError('Fermat was wrong!')
... except RuntimeError:
...   pass
...
```

(See page 110 and after for details on `try` and `raise` statements.)

# The `None` object

- A function returns `None` if it executes an expressionless `return` statement or its last statement which is not a `return` statement.
- The interactive read/parse/compile/execute/print loop skips the printing stage when expression statements return `None`.
- `None` is widely used as a default, or to indicate no choice has been made, or to indicate an exceptional condition for which raising an exception is undesired.

```
>>> None          # notice nothing is printed for this line
>>> type(None)    # None is the only member of its type
<class 'NoneType'>
>>> if None: print('yes')  # None is false
...
```

# Examples with **None**

```
>>> def safe_divide(x, y): return y != 0 and x / y or None
...
>>> safe_divide(2, 0) # notice nothing is printed for this
>>> print('<', safe_divide(2, 0), '>')
< None >
>>> safe_divide(2, 3)
0.6666666666666666
>>> x = {'a' : 1, 'b' : 2}
>>> x.get('a')
1
>>> print('<', x.get('a'), '>')
< 1 >
>>> x.get('c')          # by default, dict.get returns None
>>> print('<', x.get('c'), '>')
< None >
```

# `def` statements (functions)

- Functions can be defined using **`def`** statements.

```
>>> def fib(n):    # print Fibonacci sequence up to n
...     a, b = 0, 1
...     while b < n:
...         print (b, end=' '); a, b = b, a + b
...     # no return statement, hence returns None
...
>>> print(fib(100))
1 1 2 3 5 8 13 21 34 55 89 None
```

- Parameters (e.g., `n` in `fib` above) are always local.
- Variables assigned within a function (e.g., `a` and `b` above) are local unless declared **`global`** or **`nonlocal`**.
- Functions that only return **`None`** correspond to other languages' *procedures*.

More on `fib` in Slide 100.

# Functions are first-class objects

- Functions are objects, function names are ordinary variables, and **def** statements are a kind of assignment.

```
>>> x = fib; x
<function fib at 0x767e4b5fbc80>
>>> x(100)
1 1 2 3 5 8 13 21 34 55 89
```

- Functions can be arguments to functions.

```
>>> def f(x): return x + 3
...
>>> def g(h, y): return h(y) * 2
...
>>> g(f, 5)
16
```

# Functions are first-class objects

- Functions can be passed to, defined within, and returned from functions.

```
>>> def mul3(x): return x * 3
...
>>> def add4(x): return x + 4
...
>>> def compose(g, h):
...     def k(x): return g(h(x))
...     return k
...
>>> compose(mul3, add4)(2)
18
>>> compose(add4, mul3)(2)  # a different function!:
10
```

# Function parameters are local

Function parameters are local. Hence, assigning to a parameter does not affect the same name in an outer scope.

```
>>> def bla(l):
...     print('<before', l, '>', end=' ')
...     l = []
...     print('<after', l, '>')
...     return 9
...
>>> l = ['not', 'empty']; l
['not', 'empty']
>>> bla(l)
<before ['not', 'empty'] > <after [] >
9
>>> l                  # an l was changed, just not this one:
['not', 'empty']
```

# Objects are known by reference

- All arguments are passed to functions by value, but object values are normally references (i.e., they are is implemented as pointers).
- The internal details of a referenced object can be modified by a function and these changes persist when the function returns:

```
>>> l = ['not', 'empty']
>>> l
['not', 'empty']
>>> def exclamate(l):
...     l.append('!')
...
>>> exclamate(l)
>>> l                          # '!' added:
['not', 'empty', '!']
```

# Using outer-scope variables within functions

- If a variable x is not assigned within a function g, uses of x in g refer to the variable x of the innermost enclosing scope that defines the variable.

```
>>> def g1(): print('<x in g1:', x, '>')
...
>>> x = 9; g1()
<x in g1: 9 >
>>> def f():
...     def g2(): print('<x in g2:', x, '>')
...     x = 5; print('<x in f:', x, '>'); g2()
...
>>> print('<x globally:', x, '>'); f()
<x globally: 9 >
<x in f: 5 >
<x in g2: 5 >
```

# Modifying outer-scope variables within functions

- Normally a variable that is assigned within a function is local.
- This can be changed by declaring a variable **global** or **nonlocal**.

```
>>> def clear_l(): global l; l = []
...
>>> l = ['not', 'empty']; l; clear_l(); l
['not', 'empty']
[]
>>> def f():
...     l = ['plugh']
...     def g(): nonlocal l; l += ['xyxxy']
...     g(); return l
...
>>> f()
['plugh', 'xyxxy']
```

# `return` statements

- A `return` statement stops execution of a function and returns an object to the caller.
- If an expression is given after `return`, its value is returned, otherwise `None` is returned.
- If execution of the last statement in a function finishes, and it is not a `return` statement, the function returns `None`.

```
>>> def f(x):
...     if x < 0: return
...     if x > 0: return 'positive'
...     999
...
>>> [f(-1), f(1), f(0)]
[None, 'positive', None]
```

# Docstrings

- If the first statement in a function (**def**), **class**, or module (file) is a string literal, it is a *documentation string* (docstring) (like Common Lisp or Emacs Lisp).
- First line summarizes; details follow. (See Docstring guidelines).
- It is stored in the `__doc__` attribute where the `help` function looks.

```
>>> def my_function():
...     """Do nothing, but document it."""
...     pass
...
>>> my_function.__doc__; help(my_function)
'Do nothing, but document it.'
Help on function my_function:

my_function()
    Do nothing, but document it.
```

# `lambda` expressions (anonymous functions)

- A function whose name is only used once and whose body is a single return statement can be written with the `lambda` notation.

```
>>> def notify(n): return print('counted to', n)
...
>>> def calculate(k, callback):
...     for j in range(k):
...         k += j
...     callback(k)  # send notification of our result
...
>>> calculate(20, notify)
counted to 210
>>> calculate(20, lambda x: print('Counted To:', x))
Counted To: 210
```

# `lambda` expressions (anonymous functions)

- Here are two factories that make functions that add a fixed object to their parameter. The only difference is one uses `def` and the other `lambda`.

```
>>> def make_add_const_A(n):
...     def f(x): return x + n
...     return f
...
>>> def make_add_const_B(n):
...     return lambda x: x + n   # in SML: fn x => x+n
...
>>> f = make_add_const_A(42); g = make_add_const_B(42)
>>> [f(0), g(0)]
[42, 42]
>>> [f(5), g(5)]
[47, 47]
```

# Equality (==) and object identity (is)

Functions, modules, classes, and many other kinds of objects are compared for equality (==) by checking for identity (is). In CPython an object's identity is the address of its representation in memory.

```
>>> (lambda x: x) == (lambda x: x)  # Make 2 functions
False
>>> x = (lambda x: x); x == x       # Make 1 function
True
>>> x = [lambda x: x] * 2; x[0] == x[1]
True
```

This tells us that sequence repetition (*2) copies the reference (pointer) to (lambda x: x) rather than copying the structure.

# Exercises

- Implement Euclid's greatest common divisor algorithm as a function over 2 `int` parameters.
- Implement matrix multiplication as a function taking 2 2-dimensional matrixes as arguments. Represent each $j$ by $k$ matrix as a `list` of length $j$ containing only `list`s of length $k$.

# More `list` operations

- Read-only:
  - `l.index(x)` returns the position of `x` in `l` or raises `ValueError` if `x not in l`.
  - `l.count(x)` returns the number of occurrences of `x` in `l`.
  - `sorted(l)` returns a new list, which is the sorted version of `l`.
  - `reversed(l)` returns an iterator, which yields the elements in `l` in reverse order.
- Modifiers:
  - `l.extend(l2)` means `l[len(l) :] = l2`, i.e., add `l2` to the end of the list `l`.
  - `l.remove(x)` means `del l[l.index(x)]`, i.e., remove the first instance of `x` in `l`. Raises `ValueError` if `x not in l`.

# Usage of `lists`

- Lists can be used to model a *stack*: `append` and `pop()`.
- Lists can be used to model a *queue*: `append` und `pop(0)`.

# Higher-order functions on iterables

- `filter(test, iterable)` returns an iterator that yields each element `x` yielded by `iterable` such that `test(x)` is true.
- `map(f, iterable)` returns an iterator that yields `f(x)` for every element `x` yielded by `iterable`.

```
>>> list(filter(lambda x: x % 2 == 0, range(10)))
[0, 2, 4, 6, 8]
>>> list(map(lambda x: x * x * x, range(10)))
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
>>> # method to add 1 + 2 + 3 + 4 + 5 + 6 + ... + 9 + 10:
>>> l=list(map(lambda x,y:x+y,range(1,6),range(10,5,-1)));l
[11, 11, 11, 11, 11]
>>> sum(l)
55
```

# Higher-order functions on iterables

`reduce` in the `functools` module works on iterables in general, but here's how it works on the special case of `list`s.

- If `len(l) == 0`:
  - `reduce(f, l)` raises an exception.
  - `reduce(f, l, e)` returns `e`.
- Otherwise:
  - `reduce(f, l)` is equivalent to `reduce(f, l[1 :], l[0])`.
  - `reduce(f, l, e)` is equivalent to `reduce(f, l[1 :], f(e, l[0]))`.

```
>>> import functools
>>> 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9 / 10
2.7557319223985894e-07
>>> functools.reduce(lambda x, y: x / y, range(1, 11))
2.7557319223985894e-07
```

# `list` comprehensions

- `list` comprehensions are inspired by Haskell's list comprehensions which were inspired by mathematical *set comprehensions*.

```
>>> vec1 = [2, 4, 6]
>>> [3 * x for x in vec1]
[6, 12, 18]
>>> [3 * x for x in vec1 if x > 3]
[12, 18]
```

- The example `list` comprehensions above can be done with `map` and `filter` instead.

```
>>> list(map(lambda x: 3 * x, vec1))
[6, 12, 18]
>>> list(map(lambda x:3*x, filter(lambda y:y>3, vec1)))
[12, 18]
```

# `list` comprehensions

Some `list` comprehensions also provide the power of `functools.reduce` in addition to `map` and `filter`.

```
>>> from functools import reduce
>>> def conc(ls): return reduce(lambda c,d:c+list(d),ls,[])
...
>>> vec1 = [2, 4]; vec2 = [2, 3, -1]
>>> [(x, x**y) for x in vec1 for y in vec2]
[(2, 4), (2, 8), (2, 0.5), (4, 16), (4, 64), (4, 0.25)]
>>> conc(map(lambda x: map(lambda y:(x,x**y), vec2), vec1))
[(2, 4), (2, 8), (2, 0.5), (4, 16), (4, 64), (4, 0.25)]
>>> [x + str(y) for x in 'ab' for y in vec1]
['a2', 'a4', 'b2', 'b4']
>>> conc(map(lambda x: map(lambda y:x+str(y), vec1), 'ab'))
['a2', 'a4', 'b2', 'b4']
```

# `del` statements (deletion)

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]          # Kill first element.
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2 : 4]      # Kill 3rd + 4th elements.
>>> a
[1, 66.25, 1234.5]
>>> del a[:]          # Kill them all!!
>>> a                 # (Who says megalomania can't be fun?)
[]
>>> del a             # Now ... kill the var itself.
>>> a
NameError: name 'a' is not defined
```

# tuple objects

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> u = t, (1, 2, 3, 4, 5)    # Tuples may be nested.
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> x, y, z = t; x + y
66666
>>> empty = (); empty
()
>>> singleton = z,             # trailing comma
>>> singleton
('hello!',)
```

# `set` objects

- A set is an iterable such that iterating it will never yield the same object twice or two objects with the same hash value. The iteration order is not defined by Python.

```
>>> for x in {1, 68, 99, 68, 1, 44}: print(x, end=' ')
...
1 99 68 44
```

- Set operations: `-` (difference), `|` (union), `&` (intersection), `^` (xor).
- `x in i` tests if object `x` is an element of iterable/container `i` (except if `i` is a string when it tests if `x` is a substring of `i`).
- `set(i)` generates a `set` of the elements of the iterable `i`.
- `frozenset(i)` generates a `frozenset` (an immutable set). More on this in Slide 88.
- `list(i)` generates a `list` of the elements of the iterable `i`.
- `tuple(s)` generates a `tuple` (an immutable list).

# `set` objects

Warning: Iterating a `str` object yields its length-1 substrings (the standard Python way of representing characters).

```
>>> x = 'anteater'
>>> list(x)
['a', 'n', 't', 'e', 'a', 't', 'e', 'r']
>>> tuple(x)
('a', 'n', 't', 'e', 'a', 't', 'e', 'r')
>>> set(x)   # iteration never yields same twice (by hash)
{'a', 't', 'n', 'e', 'r'}
>>> list([x])  # maybe wanted these instead?
['anteater']
>>> tuple([x])
('anteater',)
>>> set([x])
{'anteater'}
```

# `dict` objects (dictionaries)

A `dict` object is conceptually a set of (key, value) pairs where for each key (by hash) there can be at most one value (by identity). `dict` keys must have unchanging hash values. `dict`s are also called: *finite maps/functions*, *hash maps*, *associative arrays*.

```
>>> d1 = {'jack' : 4098, 'guido' : 4127}; d1
{'jack': 4098, 'guido': 4127}
>>> d2 = dict([('guido', 4127), ('jack', 4098)]); d2
{'jack': 4098, 'guido': 4127}
>>> d1 == d2                    # equivalent
True
>>> d1['guido']                 # subscripting by key
4127
>>> d1['jack'] = 5011; d1  # a key may occur at most once
{'jack': 5011, 'guido': 4127}
```

# `dict` objects (dictionaries)

Although `dict`s are conceptually `set`s of (key, value) pairs, subscripting (`d[k]`) and iteration work differently for `dict`s than for `set`s containing pairs (which is part of why Python has both).

```
>>> d1 = {'jack' : 4098, 'guido' : 4127}; d1
{'jack': 4098, 'guido': 4127}
>>> list(d1)                          # iteration gets only keys
['jack', 'guido']
>>> s = {('jack', 4098), ('guido', 4127)}; s # set of pairs
{('guido', 4127), ('jack', 4098)}
>>> list(s)                           # iteration gets whole pairs
[('guido', 4127), ('jack', 4098)]
>>> s['jack']                         # no subscripting
TypeError: 'set' object is not subscriptable
>>> s |= {('jack', 5011)}; s   # add pair with same “key”
{('jack', 5011), ('guido', 4127), ('jack', 4098)}
```

# dict operations

```
>>> d1 = {'jack' : 4098, 'guido' : 4127}; d1
{'jack': 4098, 'guido': 4127}
>>> d1.keys()
dict_keys(['jack', 'guido'])
>>> d1.values()
dict_values([4098, 4127])
>>> d1.items()
dict_items([('jack', 4098), ('guido', 4127)])
>>> list(d1.items())    # explain why list(...)
[('jack', 4098), ('guido', 4127)]
>>> del d1['jack']; d1  # delete key once
{'guido': 4127}
>>> del d1['jack']      # delete now-non-existent key
KeyError: 'jack'
```

# Hashable objects and types

- If changing `p` can change the result of `p == q` for some `q`, then `p` is *unhashable*; otherwise `p` is *hashable*.
- See `https://docs.python.org/3/library/stdtypes.html#typesseq-mutable` onwards.
- You are expected to know that `dict` keys and `set`s and `frozenset` members must be hashable.
- `str`ings and `frozenset`s are hashable. `list`s and `set`s are unhashable. A `tuple` is hashable when all its members are.

```
>>> tel = { 'jack' : 1234 }                    # Fine
>>> tel = { frozenset(['jack']) : 1234 }       # Fine
>>> tel = { ('jack',) : 1234 }                 # Fine
>>> tel = { ['jack'] : 1234 }                  # Not fine
TypeError: unhashable type: 'list'
>>> tel = { {'jack'} : 1234 }                  # Not fine
TypeError: unhashable type: 'set'
```

# Immutable v.s mutable containers

`set` and `list` have add/append; `frozenset` and `tuple` don't:

```
>>> x = set(); x.add("*"); x
{'*'}
>>> x = frozenset(); x.add("*")
AttributeError: 'frozenset' object has no attribute 'add'
>>> x = list()        # Or just x=[]
>>> x.append("*"); x
['*']
>>> x = tuple()       # Or just x=()
>>> x.append("*")
AttributeError: 'tuple' object has no attribute 'append'
```

Warning: `{}` creates an empty `dict` (not an empty `set`). You can get an empty `set` with `set()` or `{*()}`.

# Techniques for iteration/looping

- Simultaneous iteration over positions and elements in an iterable:

```
>>> l = ['tic', 'tac', 'toe']
>>> for i, v in enumerate(l): print(i, v, end=' ')
...
0 tic 1 tac 2 toe
```

- Simultaneous iteration over two or more iterables:

```
>>> for i,v in zip(range(len(l)),l): print(i,v,end=' ')
...
0 tic 1 tac 2 toe
```

- Iteration in sorted and reversed order:

```
>>> for v in reversed(sorted(l)): print(v, end=' ')
...
toe tic tac
```

# Comparison of sequences

- Sequences are compared lexicographically, and in a nested way:

```
>>> () < ('\x00',)
True
>>> ('a', (5, 3), 'c') < ('a', (6,), 'a')
True
```

- Comparing sequences with non-sequences and comparing different types of sequences is usually an error.

```
>>> "1" < 2
TypeError: unorderable types: str() < int()
>>> () < ''
TypeError: unorderable types: tuple() < str()
>>> [0] < (0,)
TypeError: unorderable types: list() < tuple()
```

# Equality (==)

Equality is `x == y`. Seems simple—but it isn't. Equality can be defined (overloaded) for new classes to do anything we want:

```
>>> class BadEq():     # (original credit: Jared Grubb)
...     def __eq__(self, other):
...         print("Equality test alert!!!")
...         return not isinstance(other, BadEq)
...
>>> x = BadEq(); 1 == x == 'hello'  # == is not transitive!
Equality test alert!!!
Equality test alert!!!
True
>>> x == x                          # == is not reflexive!
Equality test alert!!!
False
```

Worse, the programmer can invade non-builtin classes and overwrite their `__eq__` (equality) methods. You cannot be sure what == will do.

# Equality (==) and identity (is)

The `is` operator that tests object *identity* cannot be altered.

```
>>> x = BadEq(); x is 7
False
```

This seems simple, right? No, it isn't. Functions are compared for == by address not mathematical equality of their input/output relations (cf Slide 73):

```
>>> (lambda x: x) == (lambda x: x)
False
>>> (lambda x: x) is (lambda x: x)
False
```

In a Turing-complete language it is impossible to always determine mathematical equality of input/output relations of functions and Python doesn't try (like nearly all Turing-complete languages).

# Equality (==) and identity (`is`)

Some operations copy only references to objects with complex internal structure rather than making a deep copy of the structure.

```
>>> x = [[]]
>>> y = x * 2
>>> z = [[],[]]
>>> y[0] == y[1]; y[0] is y[1]
True
True
>>> z[0] == z[1]; z[0] is z[1]  # What's happened here?:
True
False
```

`x * 2` copies only the pointer to the `[]` in the `[[]]`; it doesn't make a new empty `list`. So `y[0]` and `y[1]` are indeed *identical*.
You are expected to understand this. So for instance . . .

# Equality (==) and identity (is)

```
>>> x = [[]]; y = x * 3; y
[[], [], []]
>>> y[0].append("Ha"); y
[['Ha'], ['Ha'], ['Ha']]
>>> z = [[], [], []]; z
[[], [], []]
>>> z[0].append("Ha"); z
[['Ha'], [], []]
```

The `x * 3` operation copies the reference (pointer) to the `list` object created by `[]` inside the `list` x, so that object is shared by reference at 3 places inside `y`. Mutating it changes the printed output in all 3 places it is referenced. Every *list display* `[...]` creates a fresh, new, distinct `list` object, so `z` has three distinct `list`s within it.

# Equality (==) and identity (is)

You are expected to understand what's happening here:

```
>>> x = []
>>> y = []
>>> x == y
True
>>> x is y
False
>>> x.append("Gotcha")
>>> y
[]
```

```
>>> x = []
>>> y = x
>>> x == y
True
>>> x is y
True
>>> x.append("Gotcha")
>>> y
['Gotcha']
```

Be careful!

# copy.copy and copy.deepcopy

Python, of course, has a library for this.

```
>>> import copy
>>> l = [[]]
>>> m = copy.copy(l)
>>> n = copy.deepcopy(l)
>>> l[0].append("Hi")
>>> l
[['Hi']]
>>> m
[['Hi']]
>>> n
[[]]
```

So `copy.copy` makes a copy with links to original substructure; `copy.deepcopy` makes a 'copy' ignoring identicality . . .

# copy.copy and copy.deepcopy

. . . so let's have some innocent fun with this.

```
>>> import copy
>>> l = [[]] * 5000000      # many links to same empty list
>>> m1 = l                  # quickest possible copy
>>> m2 = copy.copy(l)       # not fast
>>> m3 = copy.deepcopy(l)   # slow, max cpu for a second
```

You are expected to anticipate what happens to `m1`, `m2`, and `m3` if we execute `l[0].append('Hi')`.

# More on recursion

You are expected to know:

- A function `f` is *recursive* when `f` calls itself (perhaps indirectly, e.g., `f` can call `g` which calls `h` which calls `f`).
- A call from function `f` to `g` is a *tail call* when it is the whole expression of a **`return`** statement and `f` has no pending unfinished business (e.g., the **`finally`** clause of a **`try`** statement).
- In such a tail call, the return value of `f` is the return value of `g` and a good compiler can reuse `f`'s stack frame, implement the call to `g` as a jump instruction, and have `g` return directly to `f`'s caller.
- *Tail-recursion* is recursion via tail calls. Good compilers turn tail recursive call cycles into fast loops, optimizing away the recursive calls and their stack allocation. (See also the SML notes.)
- Python's design does not support styles of functional programming that depend on tail-recursion being efficient.

# More on recursion

Let's look at four implementations of Fibonacci:

```
def fibi(n):  # iterative
 a, b = 1, 1
 for i in range(n):
  a, b = b, a + b
 return(a)
```

```
memo = {0:1, 1:1}
def fibm(n):  # memoised
 if not n in memo:
  memo[n]=fibm(n-1)+fibm(n-2)
 return memo[n]
```

```
def fibr(n):  # recursive
 # print(n)
 if n == 0 or n == 1:
  return(1)
 else:
  return fibr(n-1)+fibr(n-2)
```

```
def fibg(n):  # cache within
 if not hasattr(fibg, "memo"):
  fibg.memo = {0:1, 1:1}
 if not n in fibg.memo:
  fibg.memo[n] = (fibg(n-1)
                  + fibg(n-2))
 return fibg.memo[n]
```

# More on recursion

- The iterative `fibi` works fine.
- The recursive `fibr` works fine but *slowwww* because: it calls itself recursively twice and recalculates results many times (the call to `n-1` will recursively call `n-2`, again). You are expected to understand this. Uncomment `print(n)` in the `fibr` code and try `fibr(20)`.
- The memoised `fibm` is much faster. You are expected to be able to explain why.
- The memoised `fibg` is faster and uses some (non-examinable) fancy programming to keep `memo` inside the function object.

# More on recursion

The catch: Python has a global limit of 999 on recursive calls.

```
>>> fibm(10)       # works
89
>>> fibm(1010)     # doesn't work
RecursionError: maximum recursion depth exceeded
>>> fibm(50)       # works
20365011074
>>> fibm(1010)     # why does it work the second time?
8650063399098190712106206706196570348687189343891375136228333
>>> import sys; sys.getrecursionlimit()
1000
>>> sys.setrecursionlimit(40); fibr(41)
RecursionError: maximum recursion depth exceeded in comparis
>>> sys.setrecursionlimit(1000)
```

# More on recursion

- CPython needs a recursion limit because it doesn't optimise tail recursion, and the CPython designers prefer controlled crashes over uncontrolled crashes.
- Programmers have designed Python tail call optimization hacks (e.g., `http://code.activestate.com/recipes/474088/`), but it is not clear how reliable these methods are.
- Serious functional programs often reach extremely high recursion depths, e.g., processing multidimensional arrays of scientific data that have extents greater than 1 billion in multiple dimensions.
- Thus, Python has difficulty supporting some styles of functional programming.

# `import` statements (modules)

Modules normally come from Python files. The statement `import m`:

- Searches for the module `m`. This is normally done by:
  - checking the loaded module cache (`sys.modules`), or
  - checking for a built-in (written in C) module named `m`, or
  - searching each location `l` in the import path for a file at `l/m.py`, or at `l/m/__init__.py` if `m` is a *package*.
    - ★ The import path is stored in `sys.path` and initialized from the `PYTHONPATH` environment variable and system defaults. Normally the current directory is added.
    - ★ For submodules of package `p`, the attribute `p.__path__` is used instead of `sys.path`.
- If `m` is not already loaded, it is normally loaded by:
  - checking for cached compiled bytecode and reading/executing it, or
  - reading its source code, parsing it, compiling it, and executing it.
- The resulting module object is assigned to the variable `m`.

# Modules are objects

- Modules are ordinary objects.
- Members of a module can be accessed from other modules (both read and write access!) via the module object's attributes, and from within the module they can be accessed as global variables.
- The **import** statement makes an ordinary variable binding.

```
>>> import sys; s = sys; s; type(s)
<module 'sys' (built-in)>
<class 'module'>
>>> del sys; sys.stderr  # unbinds "sys" in current module
NameError: name 'sys' is not defined
>>> s.err = s.stderr # global assignment in another module!
>>> print('print errors to sys.stderr', file=s.err)
print errors to sys.stderr
```

# More on modules

- The code body of a module (file) is a suite (list) of any kind of statements except those that must be within functions (**return**, **nonlocal**, **yield**) or loops (**break**, **continue**). They are executed when the module is loaded and global definitions they make (e.g., functions, classes) become attributes of the module.
- The statement **from m import** x **as** y binds the variable y to m.x without binding the variable m, i.e., it acts quite like **import m**; y = m.x; **del** m.

```
>>> from sys import stderr as err
>>> err
<_io.StringIO object at 0x767e4bc67e58>
>>> sys
NameError: name 'sys' is not defined
>>> print('send this to the error output', file=err)
send this to the error output
```

- **WARNING:** It is usually unwise to do **from m import** *.

# Functions that convert to `string`

- `str(x)` aims for the most widely useful `string` representation of `x`.
- `repr(x)` aims for a `string` useful for debugging and ideally such that `x == eval(repr(x))`.

```
>>> s1 = '"hard" str\ting'; (str(s1), repr(s1))
('"hard" str\ting', '\'"hard" str\\ting\'')
>>> print('str: [', str(s1), '] repr: [', repr(s1), ']')
str: [ "hard" str    ing ] repr: [ '"hard" str\ting' ]
```

# string methods for formatting

```
>>> n = -123e60; n
-1.23e+62
>>> s = str(n); s
'-1.23e+62'
>>> s.rjust(13, '?') # pads on left with ?s to length 13
'????-1.23e+62'
>>> s.ljust(13)      # pads on right with spaces
'-1.23e+62    '
>>> s.center(13)     # pads on both sides
'  -1.23e+62  '
>>> s.zfill(13)      # pads with zeroes but after leading -
'-00001.23e+62'
```

# % operator on strings (formatting)

- When s is a string, the operation s % o does formatting in the style of the C printf function.

```
>>> x = 11; y = 7.5
>>> 'decimal: %d, octal: %o, float: %f' % (x, x, y)
'decimal: 11, octal: 13, float: 7.500000'
```

- The % operator also supports using dict keys to say which objects each conversion specifier formats.

```
>>> table = {'Sjoerd' : 4127, 'Jack' : 4098 }
>>> print('Jack: %(Jack)d; Sjoerd: %(Sjoerd)d' % table)
Jack: 4098; Sjoerd: 4127
```

# `try` statements (exception handling)

- A `try` statement with an `except` clause can handle exceptions.
- One `except` clause can handle several exceptions.
- A clause header "`except Exc as v:`" binds the variable `v` to the matching exception object before executing the clause body.

```
>>> try:
...     f = open('EuRpr.txt')
... except (FileNotFoundError, PermissionError,
...         IsADirectoryError) as e:
...     print("bad open: {}".format(e))
...
bad open: [Errno 2] No such file or directory: 'EuRpr.txt'
```

# `else` clauses of `try` statements

- If the `try` clause finishes normally, an `else` clause is executed.

```
>>> import sys
>>> try:
...     f = open('/dev/null')  # the always empty file
... except OSError as e:
...     print("error(%d) %s" % (e.errno, e.strerror))
... else:  # now we read the first line as an integer:
...     s = f.readline().strip()  # oops, file end gets ''
...     try:
...         i = int(s)
...     except ValueError as e2:
...         print("int failed: %s" % e2.args)
...     f.close()
...
int failed: invalid literal for int() with base 10: ''
```

# `raise` statements (exceptions)

- When a `raise` statement is executed, an exception is raised instead of continuing with the subsequent statements.
  - If `isinstance(exc, BaseException)` then `raise exc` raises the exception object `exc`.
  - If `issubclass(Exc, BaseException)` then `raise Exc` means `raise Exc()`, i.e., raise an instance of class Exc with no details.
  - `raise` by itself during execution of the body of a clause beginning with `except Exc as e:` means `raise e`, i.e., it re-raises whatever exception is currently being handled.
- When exception `e` is raised, an exception handler is sought.
  - The active call stack is searched from innermost to outermost for exception handler clauses.
  - For each clause `except h:`, the expression `h` is evaluated and the result is considered to match `e` if either `e` is an instance of class `h` or `h` is a tuple and `e` is an instance of some member of `h`.
- Control is transferred to the body of the first matching handler clause.

# Exception raising and handling example

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print ('An exception flew by!')
...     raise
...
An exception flew by!
NameError: HiThere
```

# Exception raising and handling example

```
>>> import sys
>>> try:
...     for x in range(2, -1, -1):
...         for y in range(0, 3):
...             print(y / x, end=' ')
...         print()
... except Exception as e:
...     print ("tracing:", repr(e))
...     raise    # re-raise the exception we're handling
...
0.0 0.5 1.0
0.0 1.0 2.0
tracing: ZeroDivisionError('division by zero',)
ZeroDivisionError: division by zero
```

# `finally` clauses of `try` statements

- The body of a `finally` clause will be executed at the end of the `try` statement before execution leaves the statement, no matter whether an exception is raised or handled by any of the preceding clauses.

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
```

# Exception-handling example

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(1, 2)
result is 0.5
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
```