Introduction to Python

Michael Foord
https://agileabstractions.com

- Python Trainer
- Core Python Developer
- Author of IronPython in Action
- Creator of unittest.mock
- Twitter: @voidspace

Introduction to Python

Python 3.8 or more recent is recommended for this course.

About the Course

- A hands on practical course
- Why Python and what distinguishes it from other languages
- A comprehensive foundation of the Python language
- Teaching Python for scripting, application development, DevOps, Data Science and more
- Teaching concepts of computer science alongside the practical uses
- Assumes some background in programming
- Covers the Python language not third party libraries and frameworks
- Please ask questions!

Section 1

Introduction to Python

What is Python?

- A high level, dynamically typed, object oriented, interpreted language
- Originally created by Guido Van Rossum around 1990
- Named after Monty Python
- Used widely in:
    - Teaching
    - Web application development
    - Data science
    - Scripting and Linux administration
    - Animation and GIS industries (etc)
- Python sits between shell scripting and system programming languages (like C/C++)

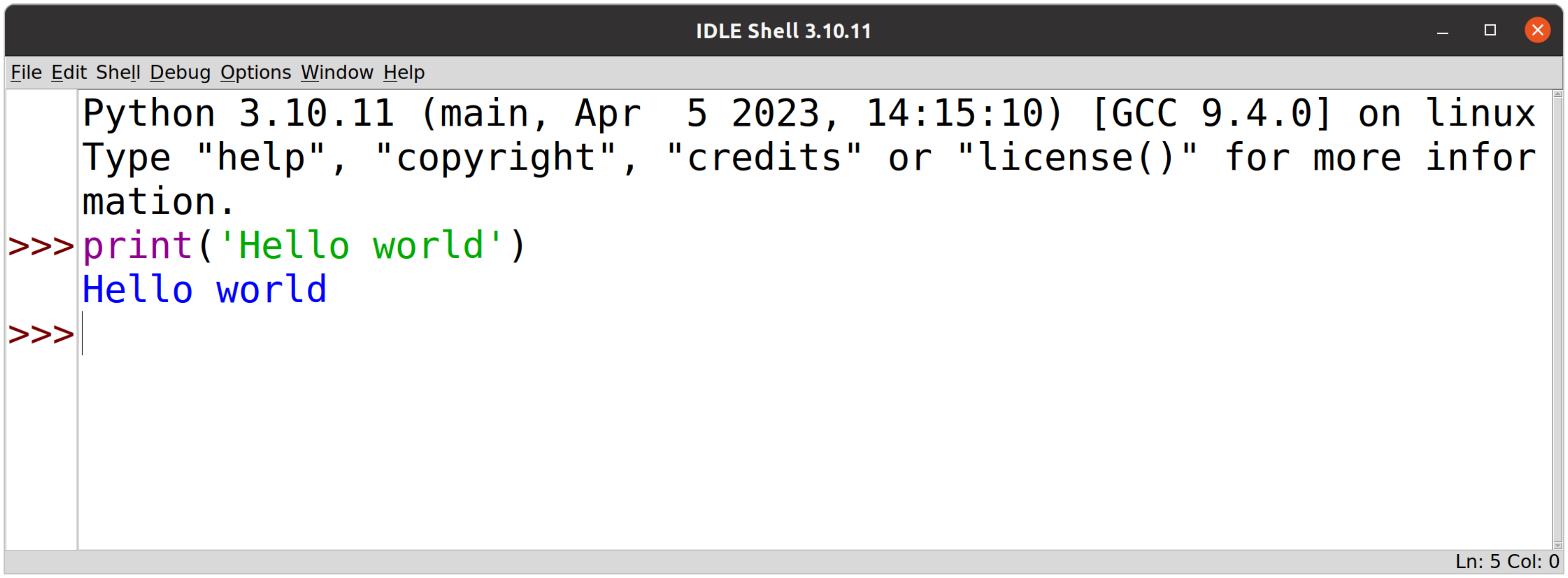
Running Python

- Python programs run inside the interpreter
- The standard interpreter can be run at the command shell/terminal (just run "python")

```
$ python
Python 3.10.11 (main, Apr  5 2023, 14:15:10) [GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- The interpreter is a valuable tool for experimentation, diagnostics and debugging

IDLE



```
Python 3.10.11 (main, Apr  5 2023, 14:15:10) [GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license()" for more infor
mation.
>>> print('Hello world')
Hello world
>>> 
```

- For this course we'll use IDLE
- IDLE is a basic code editor that comes with Python
- IDLE includes an integrated interpreter shell
- For normal development use a full IDE, like PyCharm

The Python Interpreter

- When you start Python (or IDLE) you get an interactive mode
- This is the REPL, the read-evaluate-print-loop
- Instructions you enter are evaluated immediately
- No edit/compile/run/debug cycle

```
Python 3.10.11 (main, Apr  5 2023) [GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> print('Hello World')
Hello World
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>>
```

Interactive Mode

- Some notes on using the interactive shell

>>> is the interpreter prompt for starting a new statement.

... is the interpreter prompt continuing a statement (it may appear blank in some tools).

```
>>> print('Hello World')
Hello World
>>> 37*42
1554
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>>
```

Enter a blank line to complete the statement and execute it.

Interactive Mode

- The underscore (_) variable holds the last result
- Use help() and dir() to examine objects and get information on them

```
>>> 35*7
2520
>>> _
2520
>>> help(range)

Help on class range in module builtins:

class range(object)
 |  range(stop) -> range object
 |  range(start, stop[, step]) -> range object
 |
...
```

Exercise 1.1

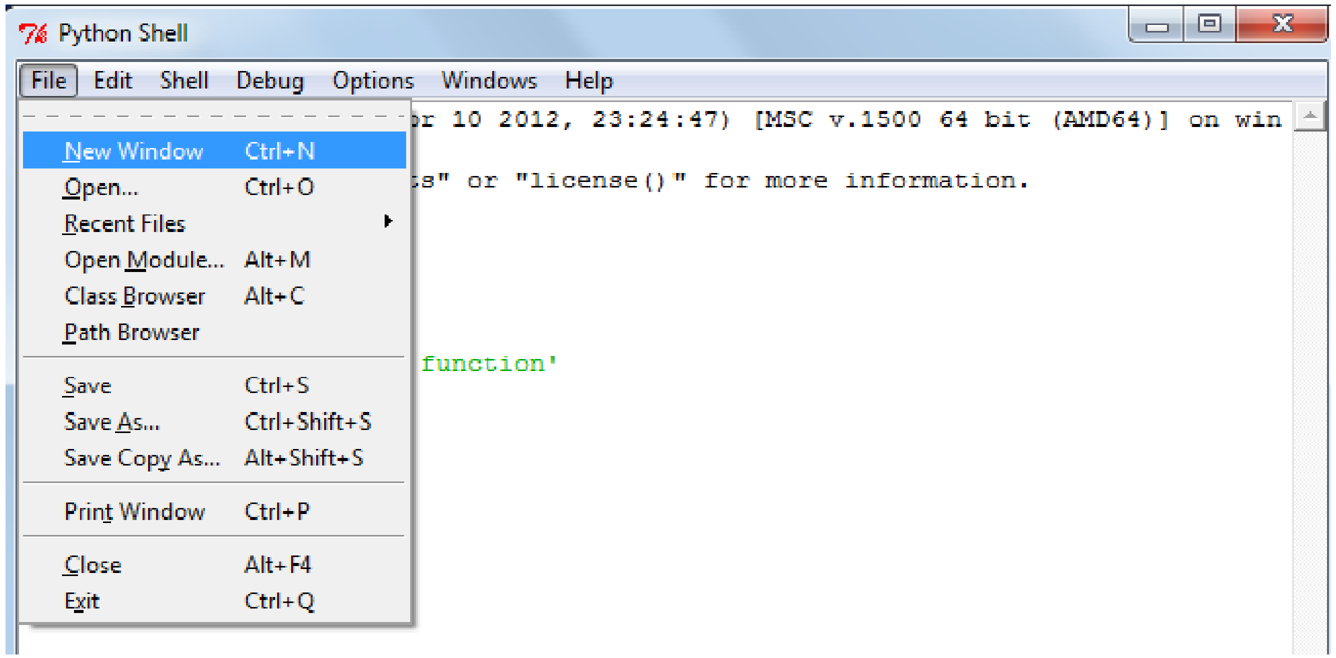
(10 minutes)

Creating Programs

- Programs are text files with the .py extension

```
# helloworld.py
print('hello world')
```

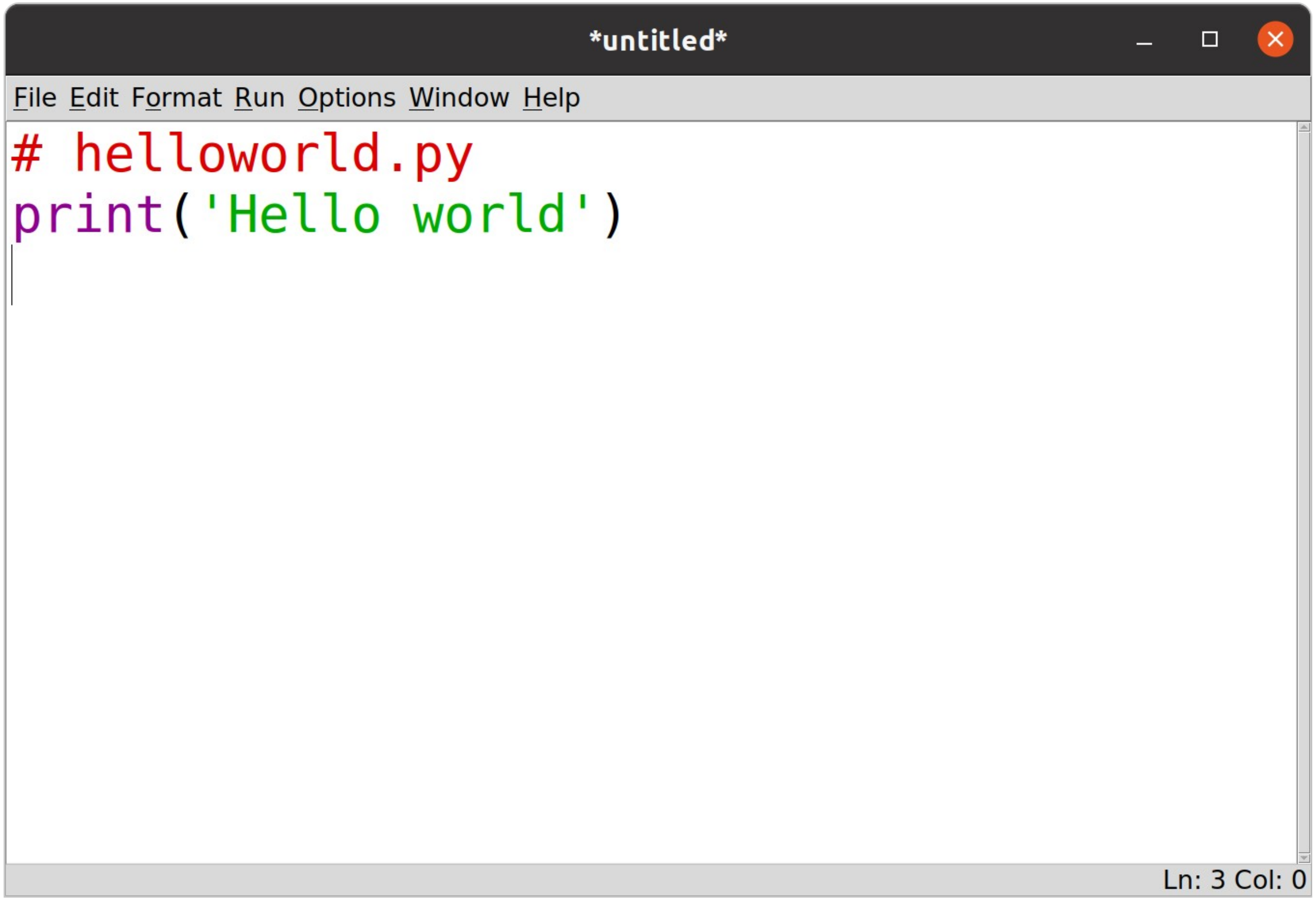
- Create them with your favourite editor
- Or the "New Window menu from IDLE



```
r 10 2012, 23:24:47) [MSC v.1500 64 bit (AMD64)] on win
s" or "license()" for more information.
 function'
```

Creating Programs

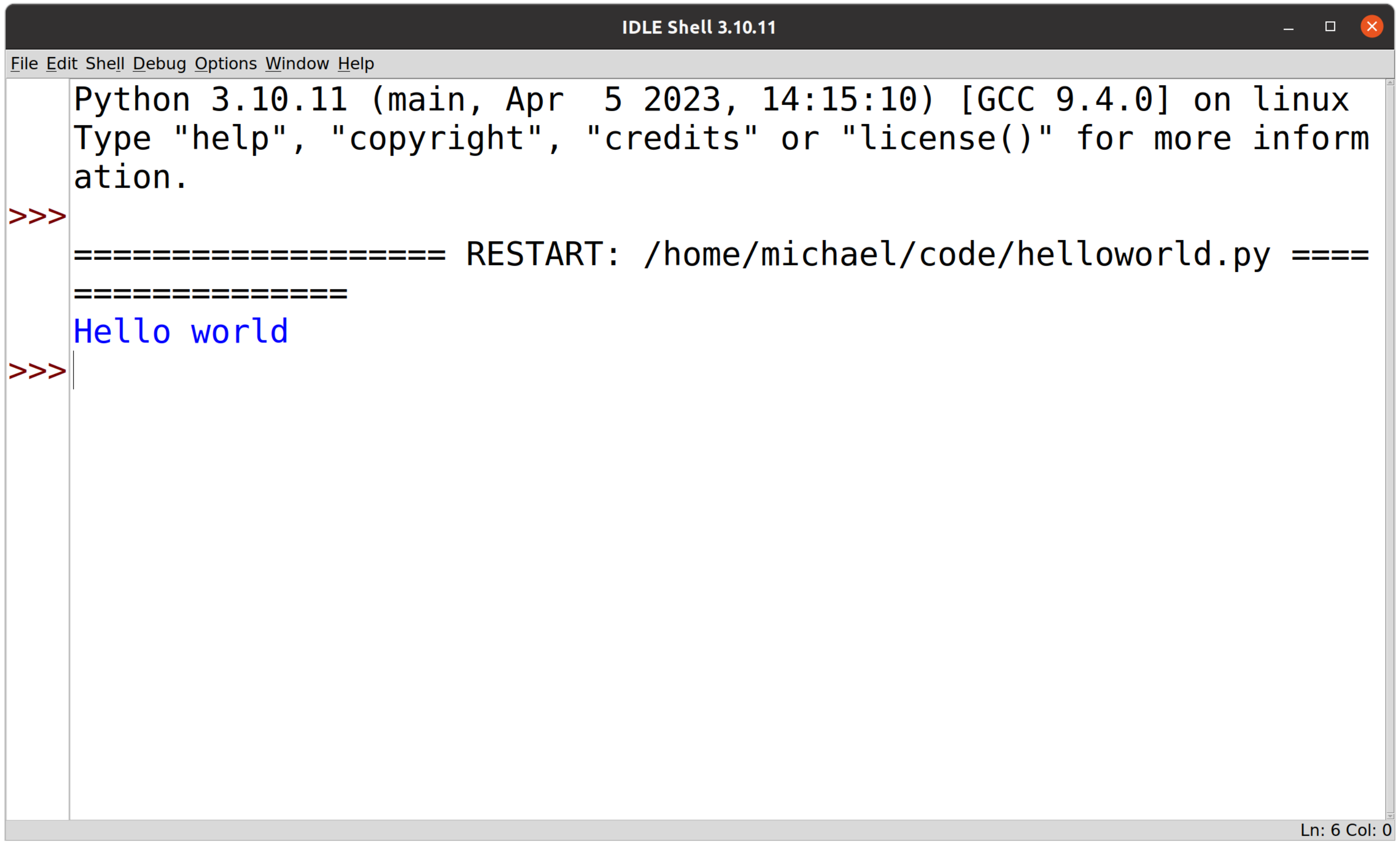
- Editing a new program in IDLE



```
# helloworld.py
print('Hello world')
```

Running Programs

- First save the file (Ctrl-S or from the menu) and then hit F5 to run the code
- Output then appears in the IDLE shell



```
Python 3.10.11 (main, Apr  5 2023, 14:15:10) [GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license()" for more inform
ation.
>>>
==================== RESTART: /home/michael/code/helloworld.py ====
==============
Hello world
>>>
```

Running Programs

- In production we're usually running programs from the command line, or on a server
- Command line on Linux/Mac:

```
$ python helloworld.py
Hello world
```

- Command shell (cmd) on Windows:

```
C:\SomeFolder>helloworld.py
hello world
C:\SomeFolder>python helloworld.py
hello world
```

A Sample Program

- The Sears Tower Problem

One morning you go out and place a dollar bill on the sidewalk by the Sears tower. Each day thereafter you go out and double the number of bills.

How many days is it before the stack of bills is taller than the Sears tower?

A Sample Program

```
# sears.py

bill_thickness = 0.11 * 0.001    # Meters (0.11 mm)
sears_height = 442               # Height (meters)
num_bills = 1
day = 1

while num_bills * bill_thickness < sears_height:
    print(day, num_bills, num_bills * bill_thickness)
    day = day + 1
    num_bills = num_bills * 2

print('Number of days', day)
print('Number of bills', num_bills)
print('Final height', num_bills * bill_thickness)
```

A Sample Program

- Output:

```
$ python sears.py
1 1 0.00011
2 2 0.00022
3 4 0.00044
4 8 0.00088
5 16 0.00176
6 32 0.00352
7 64 0.00704
...
20 524288 57.67168
21 1048576 115.34336
22 2097152 230.68672
Number of days 23
Number of bills 4194304
Final height 461.37344
```

Python 101: Statements & Comments

- A Python program is a sequence of statements
- Each statement terminated by a newline
- Statements are executed sequentially until the end of the file

- Comments are denoted by #

```
# This is a comment
height    =     442 # Meters
```

- They can be on a line of their own, or inline
- No multiline comment syntax

Python 101: Variables

- A variable is a name for a value
- The assignment statement creates a variable (a name)
- Variable names follow the same rules as most other languages [A-Za-z_][A-Za-z0-9_]*
- Unicode characters (from the character classes) are allowed, but PEP 8 specifies ascii variable names
- You do <u>not</u> declare types (int, float, etc.)

```
height = 442                # An integer
height = 442.0              # Floating point
height = 'Really tall'      # A string
```

- The types are associated with the value (the object) not the name

Python 101: Case Sensitivity

- Python is case sensitive
- These are all different variable names

```
name = 'Jake'
Name = 'Elwood'
NAME = 'Guido'
```

- Language keywords are always lowercase

```
while x < 0:        # OK
WHILE x < 0:        # ERROR
```

- So, no shouting please!

Python 101: Looping

- The while statement executes a loop (with a condition)

repeated statements

```
while num_bills * bill_thickness < sears_height:
    print(day, num_bills, num_bills * bill_thickness)
    day = day + 1
    num_bills = num_bills * 2

print('Number of days', day)
```

- Executes the statements indented underneath, while the condition is true

Python 101: Indentation

- Indentation must be consistent

```
while num_bills * bill_thickness < sears_height:
    print(day, num_bills, num_bills * bill_thickness)
        day = day + 1
    num_bills = num_bills * 2

print('Number of days', day)
```

(error)

- A colon indicates the start of a new block

```
while num_bills * bill_thickness < sears_height:
```

Python 101: Indentation

- There is a standard indentation style
  - Always use spaces
  - Use 4 spaces per level of indentation
  - Avoid tabs (most editors will convert to spaces)

- Always use a Python aware code editor

Python 101: Conditionals

- If

```
if a < b:
    print('Computer says no')
```

- If-else

```
if a < b:
    print('Computer says no')
else:
    print('Computer says yes')
```

- If-elif-else

```
if a < b:
    print('Computer says no')
elif a > b:
    print('Computer says yes')
else:
    print('Computer says maybe')
```

Python 101: Printing

- The print function

```
print(x)
print(x, y, z)
print('Your name is', name)
print(x, end=' ')              # Omits newline
```

- Produces a single line of text
- Items are separated by a space
- Always prints a newline unless the optional end argument is supplied

Python 101: User Input

- To read a line of user typed input:

```
name = input('Enter your name:')
```

- Prints a prompt, returns the typed response
- Useful for command line tools and simple programs, and sometimes for debugging/diagnostics
- It's not widely used for real programs (we'll rarely use it in this course)

Python 101: The pass Statement

- Sometimes you need to specify an empty block of code (like {} in C/Java)
- pass is the no-op statement (it does nothing)
- Possibly as a placeholder for code to be implemented later

```
if name in namelist:
    # Not implemented yet
    pass
else:
    statements
```

- Or where you need some code but there's nothing to do

```
class MyError(Exception):
    pass
```

Exercise 1.2

(10 minutes)

Numbers

- Python has four types of numbers

  - Booleans
  - Integers
  - Floating point
  - Complex (imaginary numbers)

Booleans

- Two values: True, False

```
a = True
b = False
```

- Technically a subclass of integer (inherited from C)
- Evaluated with values 1, 0

```
c = 4 + True    # c = 5
d = False
if d == 0:
    print('d is False')
```

- Never do this in practise!

Integers

- Signed values of arbitrary size (only limited by memory!)

```
a = 37
b = 1_000_000
c = -299392993727716627377128481812241231
d = 0x7fa8      # Hexadecimal
e = 0o253       # Octal
f = 0b10001111  # Binary
```

- Common operations

+	Add	<<	Bit shift left
-	Subtract	>>	Bit shift right
*	Multiply	&	Bit-wise AND
/	Divide (a float)	|	Bit-wise OR
//	Floor divide	^	Bit-wise XOR
%	Modulo	~	Bit-wise NOT
**	Power	abs(x)	Absolute value

Comparisons

- Comparison/relational operators

```
< > <= >= == !=
```

- Boolean expressions (and, or, not)

```
if b >= a and b <= c:
    print('b is between a and c')
if not (b < a or b > c):
    print('b is still between a and c')
```

Floating Point (float)

- Use a decimal or exponential notation

```
a = 37.45
b = 4e5
c = -1.345e-10
```

- Represented as "double precision" (64bit) using the native CPU representation, following the IEEE 754 spec

```
17 digits of precision
Exponent from -308 to 308
```

- The same as the C double type (and in most other languages)

Floating Point

- Beware that floating point numbers are inexact

```
>>> a = 2.1 + 4.2
>>> a == 6.3
False
>>> a
6.300000000000001
```

- This is not specific to Python, it's how floating point numbers work
- Floats only have 17 bits of precision
- The results of calculations may not be exactly what you expect (not a Python bug!)

Floating Point Operators

`+`	Add
`-`	Subtract
`*`	Multiply
`/`	Divide
`%`	Modulo
`**`	Power
`abs(x)`	Absolute value

- Additional functions are in the math module

```
import math
a = math.sqrt(x)
b = math.sin(x)
c = math.cos(x)
d = math.tan(x)
e = math.log(x)
```

Converting Numbers

- Type names can be used to convert

```
a = int(x)          # Convert x to integer
b = float(x)        # Convert x to float
```

- Example:

```
>>> a = 3.14159
>>> int(a)
3
```

- Also works with strings containing numbers:

```
>>> a = '3.14159'
>>> float(a)
3.14159
```

Exercise 1.3

(15 minutes)

Strings

- Literals written with quotes (no difference in result between single and double quotes)

```
a = 'Yeah but no but yeah but...'
b = "computer says no"
c = '''
Look into my eyes, look into my eyes,
the eyes, the eyes, the eyes,
not around the eyes,
don't look around the eyes,
look into my eyes, you're under.
'''
```

- The standard escape codes work (e.g. '\n', '\t')
- Triple quotes capture all the text, including newlines, between them

String Escape Codes

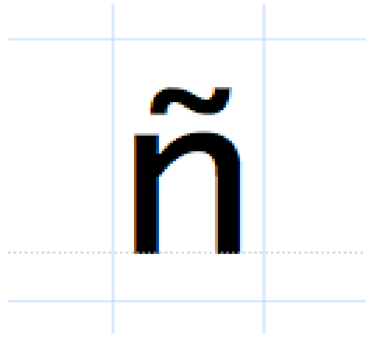
- Standard escape codes work

`'\n'`	Line feed
`'\r'`	Carriage return
`'\t'`	Tab
`'\''`	Single quote
`'\"'`	Double quote
`'\\'`	Backslash

- The codes are inspired by C/Unix

String Representation

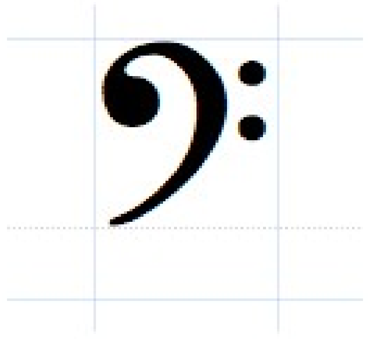
- Strings are a sequence of unicode code points


LATIN SMALL LETTER N WITH TILDE

Unicode U+00F1

FOR ALL

Unicode U+2200


MUSICAL SYMBOL F CLEF

Unicode U+1D122 (U+D834 U+DD22)

```
a = '\xf1'            # a = 'ñ'
b = '\u2200'          # b = '∀'
c = '\U0001D122'      # c = ''
d = '\N{FOR ALL}'     # d = '∀'
```

String Representation

- Strings are sequences and can be indexed: s[n]

```
a = 'Hello world'
b = a[0]            # b = 'H'
c = a[4]            # c = 'o'
d = a[-1]           # d = 'd' (taken from the end of the string)
```

- Slicing/sub-strings: s[start:end]

```
d = a[:5]       # d = 'Hello'
e = a[6:]       # e = 'world'
f = a[3:8]      # f = 'lo wo'
g = a[-5:]      # g = 'world'
```

- Concatenation: +

```
a = 'Hello' + 'World'
b = 'Say ' + a
```

More String Operations

- Length (len)

```
>>> s = 'Hello'
>>> len(s)
5
```

- Membership test (in, not in)

```
>>> 'e' in s
True
>>> 'x' in s
False
>>> 'hi' not in s
True
```

- Replication (s*n)

```
>>> s = 'Hello'
>>> s*5
'HelloHelloHelloHelloHello'
```

String Methods

- Strings have "methods" that perform various operations with the string data

- Strip leading and trailing whitespace

```
t = s.strip()
```

- Case conversion

```
t = s.lower()
t = s.upper()
```

- Replacing text

```
t = s.replace('Hello', 'Hallo')
```

More String Methods

```
s.endswith(suffix)           # Check if string ends with suffix
s.find(t)                    # First occurrence of t in s
s.index(t)                # First occurrence of t in s
s.isalpha()                  # Check if characters are alphabetic
s.isdigit()                  # Check if characters are numeric
s.islower()                  # Check if characters  are lowercase
s.isupper()                  # Check if characters are uppercase
s.join(slist)                # Join strings using s as delimiter
s.lower()                    # Convert to lowercase
s.replace(old,new)           # Replace text
s.rfind(t)                # Search for t from end of string
s.rindex(t)                  # Search for t from end of string
s.split([delim])             # Split s into list of substrings
s.startswith(prefix)         # Check if string starts with prefix
s.strip()                    # Strip leading/trailing whitespace
s.upper()                    # Convert to uppercase
```

- Consult the documentation for the gory details

String Mutability

- Strings are immutable (read only)
- Once created the value can't be changed

```
>>> s = 'Hello world'
>>> s[0] = 'h'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

- All operations and methods that manipulate string data always create new strings

String Conversions

- Use str() to convert values into strings

```
>>> x = 42
>>> str(x)
'42'
```

- The resulting text is the same as produced by print

Byte Strings

- A string of 8-bit bytes
- Byte strings use the b string prefix

```
data = b'Hello World\r\n'
```

- Most of the usual string operations work

```
len(data)                                   # -> 13
data[0:5]                                   # -> b'Hello'
data.replace(b'Hello', b'Cruel')            # -> b'Cruel World\r\n'
```

- Indexing is a bit different (returns integer byte values)

```
data[0]    # -> 72 (ASCII code for 'H')
```

- Conversion to/from text (Unicode ↔ bytes)

```
text = data.decode('utf-8')   # bytes -> text
data = text.encode('utf-8')   # text -> bytes
```

Raw Strings

- Strings with an uninterpreted backslash
- Raw strings use the r prefix

```
r'c:\newdata\test'
```

- String is the literal text exactly as typed
- Used where the backslash (\) has special significance
- Especially useful for regular expressions and Windows file paths

f-Strings

- Strings with formatted expression substitution
- f-strings use the f prefix and are evaluated immediately

```
name = 'IBM'
shares = 100
price = 91.1
```

```
>>> f'We hold {shares} shares of {name}.'
'We hold 100 shares of IBM.'
>>> f'{name:>10s} {shares:10d} {price:10.2f}'
'       IBM        100      91.10'
>>> f'Cost = ${shares*price:0.2f}'
'Cost = $9110.00'
>>> f'{name=}, {shares=}, {price=}'   # New in Python 3.8
"name='IBM', shares=100, price=91.1"
>>> f'Expressions in {{braces}} are evaluated: e.g. {name.lower()}'
'Expressions in {braces} are evaluated: e.g. ibm'
```

Exercise 1.4

(10 minutes)

String Splitting

- Strings often represent fields of data
- To work with each field we split the string into a listen

```
>>> line = 'GOOG,100,490.10'
>>> row = line.split(',')
>>> row
['GOOG', '100', '490.10']
```

- Example: When reading row oriented data from a file, like a csv file, we might read each line and then split the line into columns
- The string split() method takes a delimiter and returns a list of components

Lists

- A sequence of arbitrary values

```python
names = ['Elwood', 'Jake', 'Curtis']
nums = [39, 38, 42, 65, 111]
```

- Adding new items (append, insert)

```python
names.append('Murphy')          # Adds at end
names.insert(2, 'Aretha')       # Inserts in middle
```

- Concatenation: s + t

```python
s = [1, 2, 3]
t = ['a', 'b']

s + t   ──────────────────────►  [1, 2, 3, 'a', 'b']
```

Lists

- Lists are indexed by integers, starting at 0

```
names = [ 'Elwood', 'Jake', 'Curtis' ]

names[0] ──────────────────► 'Elwood'
names[1] ──────────────────► 'Jake'
names[2] ──────────────────► 'Curtis'
```

- Negative indices are from the end

```
names[-1] ──────────────────►  'Curtis'
```

- Lists are mutable, the contents can be changed
- Changing an item at an index position

```
names[1] = 'Joliet Jake'
```

More List Operations

- Length (len)

```
>>> names = ['Elwood','Jake','Curtis']
>>> len(names)
3
```

- Membership test (in, not in)

```
>>> 'Elwood' in names
True
>>> 'Britney' not in names
True
```

- Replication (s * n)

```
>>> s = [1, 2, 3]
>>> s * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

List Iteration & Search

- Iterating over the list contents

```
for name in names:
    # use name
    ...
```

- name is the iteration variable, it gets a new value every time through the loop
- Similar to the 'foreach' statement from other programming languages

- To find the position of a value, use the index method

```
>>> names = ['Elwood','Jake','Curtis']
>>> names.index('Curtis')
2
>>> names[2]
'Curtis'
```

List Removal

- Removing an item by value

```python
names.remove('Curtis')
```

- Deleting an item by index

```python
del names[2]
```

- Removal results in items moving down to fill the space vacated (no holes)

List Sorting

- Lists can be sorted "in-place" with the sort method

```
s = [10, 1, 7, 3]
s.sort()                          # s = [1, 3, 7, 10]
```

- Sorting in reverse order

```
s = [10, 1, 7, 3]
s.sort(reverse=True)              # s = [10, 7, 3, 1]
```

- Sorting works with any ordered (comparable) data

```
s = ['foo', 'bar', 'spam']
s.sort()                          # s = ['bar', 'foo', 'spam']
```

- You can't sort lists of mixed types (incomparable)

```
>>> a = [None, 1, 5, 0, 99]
>>> a.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and
'NoneType'
```

Lists and Maths

- Caution: lists are not designed for maths

```
>>> nums = [1, 2, 3, 4, 5]
>>> nums * 2
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
>>> nums + [10, 11, 12, 13, 14]
[1, 2, 3, 4, 5, 10, 11, 12, 13, 14]
```

- Lists are not vectors or matrices
- Not the same as in Matlab, Octave, IDL, etc
- For vector or matrix operations use third party libraries like numpy or Pandas

Exercise 1.5

(10 minutes)

File Input and Output

- Opening a file

```
f = open('foo.txt','rt')                    # Open a file for reading
g = open('bar.txt','wt')                    # Open a file for writing
```

- To read data

```
data = f.read([maxbytes])       # Read up to maxbytes bytes
```

- To write text to a file

```
g.write('some text')
```

- Closing the file handle when you're done

```
f.close()
```

File Management

- File handles are an operating system resource and must be closed when you're done with them

```
f = open(filename, 'rt')
# Use the file f
...
f.close()
```

- For production code always use the 'with' statement (using the file handle as a 'context manager')

```
with open(filename, 'rt') as f:
    # Use the file f
    ...
                   ←———————— file f is closed here
statements
```

- The file is automatically closed when control leaves the indented block

Reading File Data

- Reading an entire file as a string

```
with open(filename, 'rt') as f:
    data = f.read()
```

- Processing a file a line at a time by iterating over the file handle

```
with open(filename, 'rt') as f:
    for line in f:
        # Process the line
        ...
```

Writing to a File

- Writing string data

```
with open('outfile', 'wt') as f:
    f.write('Hello World\n')
    ...
```

- Redirecting the print function

```
with open('outfile', 'wt') as f:
    print('Hello World', file=f)
    ...
```

Exercise 1.6

(15 minutes)

Simple Functions

- Use functions for code you want to reuse

```python
def sumcount(n):
    '''
    Returns the sum of the first n integers
    '''
    total = 0
    while n > 0:
        total += n
        n -= 1
    return total
```

- Calling a function

```python
a = sumcount(100)
```

- A function is a series of statements that performs a task and returns a result

Library Functions

- Python comes with a large standard library ("batteries included")
- Library modules are accessed with the import statement

```
import math
x = math.sqrt(10)

import urllib.request
u = urllib.request.urlopen('https://www.python.org/')
data = u.read()
```

- We'll cover some interesting parts of the standard library as we go and cover functions in more detail shortly

Exception Handling

- Errors are reported as exceptions
- Unhandled exceptions cause the program to stop and a traceback is printed

```
>>> int('N/A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'N/A'
```

- For debugging the traceback shows you the error type, a message describing the exception, and whereabouts in the code it happened

Exceptions

- Exceptions can be caught and handled
- To handle them use the try...except statement

```
for line in f:
    fields = line.split()
    try:
        shares = int(fields[1])
    except ValueError:
        print("Couldn't parse", line)
    ...
```

The type (name) must match the error you're trying to catch

```
>>> int('N/A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'N/A'
```

Exceptions

- To raise an exception, use the raise statement

```python
raise RuntimeError('What a kerfuffle')
```

- Unless the exception is caught by a try...except it will cause the program to terminate with a non-zero error code and a traceback

```
$ python foo.py
Traceback (most recent call last):
  File "/home/michael/code/foo.py", line 1, in <module>
    raise RuntimeError('What a kerfuffle')
RuntimeError: What a kerfuffle
```

Summary

- This has been an overview of simple Python
- Enough to write basic programs
- We've covered some of the core datatypes and language constructs (loops, conditions, files, etc)

Exercise 1.7

(15 minutes)

Section 2

Working with Data

Overview

- Most programs work with data
- In this section we look at how Python programmers work with and represent data
- Common programming idioms
- How to (not) shoot yourself in the foot

Primitive Datatypes

- Python has a few primitive types of data
  - Integers
  - Floating point
  - Strings (text)
- Obviously all programs use these

None Type

- Nothing, nil, null, nada

```
email_address = None
```

- None is often used as a placeholder for optional or missing values

```
if email_address is not None:
    send_email(email_address, msg)
```

Data Structures

- Real programs have more complex data
- Example: a holding of a stock

```
100 shares of GOOG at $490.10
```

- An "object" with three parts
  - Name ("GOOG", a string)
  - Number of shares (100, an integer)
  - Price (490.10, a float)

Tuples

- A collection of values grouped together
- Example:

```python
s = ('GOOG', 100, 490.1)
```

- Sometimes the () are omitted in the syntax

```python
s = 'GOOG', 100, 490.1
```

- Special cases (0-tuple, 1-tuple)

```python
t = ()
w = ('GOOG',)
```

Tuple Use

- Tuples are usually used to represent simple records or structures

```
contact = ('Michael Foord', 'michael@python.org')
stock = ('GOOG', 100, 490.1)
host = ('www.python.org', 80)
```

- A single "object" of multiple parts
- Like a row in a csv file, or a database, every position (index) has meaning

Note: Tuple use is typically different from a list. Lists are normally used where every entry is a distinct item, usually of the same type.

```
names = ['Elwood', 'Jake', 'Curtis']
```

Tuples

- Tuples are sequences, they are ordered

```
s = ('GOOG',100, 490.1)
name = s[0]         # 'GOOG'
shares = s[1]       # 100
price = s[2]        # 490.1
```

- Tuples are immutable, the contents can't be modified

```
>>> s = ('GOOG',100, 490.1)
>>> s[1] = 150
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

- You can however make a new tuple

```
s2 = (s[0], 150, s[2])
```

Tuple Packing

- Creating a tuple is called a "packing" operation
- Several items are packed together as a single entity

```
s = ('GOOG', 100, 490.1)
```

- The tuple is then easy to pass around to other parts of a program as a single object

Tuple Unpacking

- To use the tuple you can unpack the its parts into variables

```
name, shares, price = s

print(f"Cost = {shares * price}")
```

- Number of variables must match the tuple structure

```
>>> name, shares = s  # ERROR
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

Dictionaries

- A hash table or associative array
- A collection of values indexed by "keys"
- The keys serve as field names

```
s = {
    'name': 'GOOG',
    'shares': 100,
    'price': 490.1
}
```

Dictionaries

- Getting values: use the key name

```
>>> print(s['name'], s['shares'])
GOOG 100
>>> s['price']
490.1
```

- Adding/modifying values: assign to key names

```
>>> s['shares'] = 150
>>> s['date'] = '2023-05-22'
```

- Deleting a value

```
>>> del s['date']
```

Dictionaries

- Dictionaries are useful when
  - There are <u>many</u> different values
  - The values will be modified/manipulated
- Dictionaries can improve code clarity (but tuples are faster and use slightly less memory)

```
s['price']      vs        s[2]
```

Exercise 2.1

(10 minutes)

Containers

- Programs often have to work with many objects
  - A portfolio of stocks
  - A table of stock prices
- Three (primary) choices:
  - Lists (ordered data)
  - Dictionaries (unordered data, accessed by key)
  - Sets (unordered collection)

Lists as a Container

- Use a list when the order of the data matters (or for processing sequentially)
- Lists can hold any kind of object
- Example: a list of tuples

```
portfolio = [
    ('GOOG', 100, 490.1),
    ('IBM', 50, 91.3),
    ('CAT', 150, 83.44)
]

portfolio[0]  ──────────────────────►  ('GOOG', 100, 490.1)
portfolio[1]  ──────────────────────►  ('IBM', 50, 91.3)
```

List Construction

- Example of building a list from scratch

```
records = []    # Initial empty list

# Use .append() to add more items
records.append(('GOOG', 100, 490.10))
records.append(('IBM', 50, 91.3))
...
```

- Example: reading records from a file

```
records = []        # Initial empty list

with open('portfolio.csv', 'rt') as f:
    for line in f:
        row = line.split(',')
        stock = (row[0], int(row[1])), float(row[2])
        records.append(stock)
```

Dicts as a Container

- Dicts are useful if you want fast random lookups (by keyname)
- Example: a dictionary of stock prices (a lookup table)

```
prices = {
    'GOOG': 513.25,
    'CAT': 87.22,
    'IBM': 93.37,
    'MSFT': 44.12
    ...
}

>>> prices['IBM']
93.37
>>> prices['GOOG']
513.25
```

Dict Construction

- Example of building a dict from scratch

```
prices = {}     # Initial empty dict

# Insert new items
prices['GOOG'] = 513.25
prices['CAT'] = 87.22
prices['IBM'] = 93.37
```

- Example: populating from a file

```
prices = {}     # Initial empty dict

with open('prices.csv', 'rt') as f:
    for line in f:
        row = line.split(',')
        name = row[0]
        price = float(row[1])
        prices[name] = price
```

Dictionary Lookups

- To test for the existence of a key

```
if key in d:
    # Yes
else:
    # No
```

- Looking up a value that might not exist

```
name = d.get(key, default)
```

- Example:

```
>>> prices.get('IBM', 0.0)
93.37
>>> prices.get('SCOX', 0.0)
0.0
```

Composite Keys

- Use tuples

```
holidays = {
    (1, 1) : 'New Years',
    (3, 14) : 'Pi day',
    (9, 13) : "Programmer's day",
}
```

- Access

```
>>> holidays[3, 14]
'Pi day'
```

Sets

- Sets (a "bag")

```
tech_stocks = { 'IBM','AAPL','MSFT' }
tech_stocks = set(['IBM', 'AAPL', 'MSFT'])
```

- A collection of unordered unique items
- Built on a hash table
- Useful for keeping track of things and for membership tests

```
>>> 'IBM' in tech_stocks
True
>>> 'FB' in tech_stocks
False
```

Sets

- Sets are useful for duplicate elimination

```
names = ['IBM', 'AAPL', 'GOOG', 'IBM', 'GOOG', 'YHOO']
unique = set(names)
# unique = set(['IBM', 'AAPL','GOOG','YHOO'])
```

- Other set operations

```
names.add('CAT')        # Add an item
names.remove('YHOO')    # Remove an item

s1 | s2     # Set union
s1 & s2     # Set intersection
s1 - s2     # Set difference
```

Exercise 2.2

(30 minutes)

Formatted Output

- When working with data you often want to produce structured output (tables, etc)

```
      Name     Shares      Price
---------- ---------- ----------
        AA        100      $9.22
       IBM         50    $106.28
       CAT        150     $35.46
      MSFT        200     $20.89
        GE         95     $13.48
      MSFT         50     $20.89
       IBM        100    $106.28
```

f-strings

- f-strings

```
name = 'IBM'
shares = 100
price = 91.1

>>> f'{name:>10s} {shares:>10d} {price:>10.2f}'
'       IBM        100      91.10'
```

- *{expr:fmt}* within the string is replaced
- Commonly used with print

```
print(f'{name:>10s} {shares:>10d} {price:>10.2f}')
```

- Variables (new in Python 3.8):

```
>>> f'{name=}, {shares=}, {price=}'
"name='IBM', shares=100, price=91.1"
```

Format Codes

d	Decimal integer
b	Binary integer
x	Hexadecimal integer
f	Float as [-]m.dddddd
e	Float in exponential format [-]m.ddddd e+/-xx
g	Float but selective use of the e notation
s	String
c	Character (from integer)

Modifiers (partial list)

:>10d	Integer, right-aligned to 10 spaces
:<10d	Integer, left-aligned to 10 spaces
:*>10d	Integer, right-aligned, padded with *
:^10d	Integer, centre-aligned to 10 spaces
:0.2f	Float with 2 decimal spaces

String Format Methods

- Strings have format and format_map methods
- Strings become reusable templates
- Uses the same formatting mini language
- format works by position or keyword arguments

```
>>> template = '{:>10s} {:>10d} {:10.2f}'
>>> template.format(name, shares, price)
'       IBM        100      91.10'
>>> template = '{name:>10s} {shares:>10d} {price:10.2f}'
>>> template.format(name=name, shares=shares, price=price)
'       IBM        100      91.10'
```

- format_map takes a dictionary

```
>>> stock = {'name': name, 'shares': shares, 'price': price}
>>> template.format_map(stock)
'       IBM        100      91.10'
```

Exercise 2.3

(20 minutes)

Working with Sequences

- Python has three core "sequence" datatypes

```
a = 'Hello'                       # String
b = [1, 4, 5]                     # List
c = ('GOOG', 100, 490.1)          # Tuple
```

- Sequences are ordered: s[*n*]

```
a[0]    ──────────► 'H'
b[-1]   ──────────► 5
c[1]    ──────────► 100
```

- Sequences have a length: len(s)

```
len(a) ──────────► 5
len(b) ──────────► 3
len(c) ──────────► 3
```

Working with Sequences

- Sequences can be replicated: s * n

```
>>> a = 'Hello'
>>> a * 3
'HelloHelloHello'
>>> b = [1, 2, 3]
>>> b * 2
[1, 2, 3, 1, 2, 3]
```

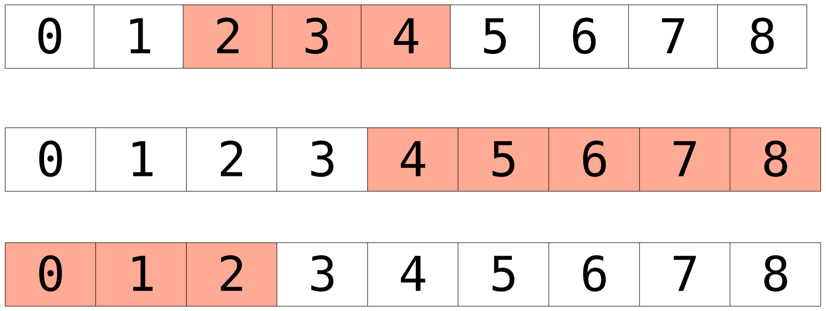
- Similar sequences can be concatenated: s + t

```
>>> a = (1, 2, 3)
>>> b = (4, 5)
>>> a + b
(1, 2, 3, 4, 5)
```

Sequence Slicing

- Slicing operator: `s[`*`start`*`:`*`end`*`]`

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

a[2:5]  ──────────→ [2, 3, 4]

a[-5:]  ──────────→ [4, 5, 6, 7, 8]

a[:3]   ──────────→ [0, 1, 2]
```



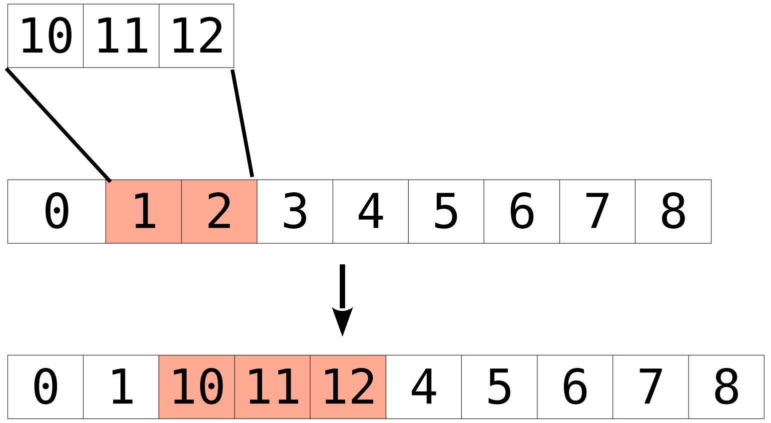
- Indices must be integers
- Slices do <u>not</u> include end values
- If indices are omitted they default to the start or the end of the sequence
- Handy way to copy a sequence:

```
a[:]
```

Additional Slicing

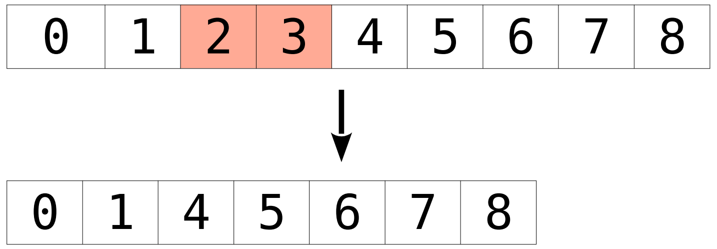
- Slice reassignment

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

a[2:4] = [10, 11, 12]
```



- Slice deletion

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

del a[2:4]
```



Sequence Reductions

- sum(s)

```
>>> s = [1, 2, 3, 4]
>>> sum(s)
10
```

- min(s), max(s)

```
>>> t = ['Hello', 'World']
>>> min(s)
1
>>> max(s)
4
>>> max(t)
'World'
```

Iterating Over a Sequence

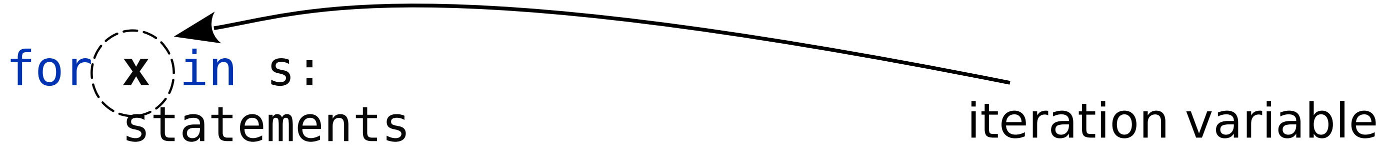
- The for loop iterates over sequence database

```
>>> s = [1, 4, 9, 16]
>>> for i in s:
...     print(i)
...
1
4
9
16
```

- On each iteration of the loop you get a new item of data to work with
- You can iterate over any "iterable" not just sequences

Iteration Variables

- Each time through the loop a new value is placed in the iteration variable



```
for x in s:
    statements
```

- Overwrites the previous value (if any)
- After the loop the variable retains the last value

break statement

- Breaking out of a loop (exiting)

```
for name in namelist:
    if name == username:
        break
    ...
    ...
statements
```

- Only applies to the innermost list

continue statement

- Skipping to the next iteration

```
for line in lines:
    if line == '\n':                    # Skip blank lines
        continue
    # More statements
    ...
```

- Useful if the current item isn't of interest or needs to be ignored in processing

Looping Over Integers

- If you need to count, use range()
- range([*start*,] *end* [,*stop*])

```
for i in range(100):
    # i = 0, 1,..., 99
for j in range(10, 20):
    # j = 10, 11,..., 19
for k in range(10, 50, 2):
    # k = 10, 12,..., 48
```

- Note: the end value is never included (the same behaviour as slicing)

enumerate() function

- enumerate(*sequence* [, *start=0*])
- Provides a loop counter

```
names = ['Elwood', 'Jake', 'Curtis']
for i, name in enumerate(names):
    # Loops with i = 0, name = 'Elwood'
    #            i = 1, name = 'Jake'
    #            i = 2, name = 'Curtis'
    ...
```

- Example: keeping track of line number

```
with open(filename) as f:
    for lineno, line in enumerate(f, start=1):
        ...
```

enumerate() function

- enumerate() is a nice shortcut

```
for i, x in enumerate(s):
    statements
```

- Compare to:

```
i = 0
for x in s:
    statements
    i += 1
```

- Less typing and enumerate() runs slightly faster

for and tuples

- You can have multiple iteration variables

```
points = [
    (1, 4),(10, 40),(23, 14),(5, 6),(7, 8)
]

for x, y in points:
    # Loops with x = 1, y = 4
    #            x = 10, y = 40
    #            x = 23, y = 14
    #            ...
```

tuples are exanded

- Here each tuple is <u>unpacked</u> into a set of iteration variables

zip() function

- Makes an iterator that combines sequences

```
columns = ['name', 'shares', 'price']
values = ['GOOG', 100, 490.1 ]

pairs = zip(a, b)
# ('name','GOOG'), ('shares',100), ('price',490.1)
```

- To get the result, you must iterate

```
for name, value in pairs:
    ...
```

- Common use: making dictionaries

```
d = dict(zip(columns, values))
```

Exercise 2.4

(15 minutes)

collections module

- Contains several other useful container types
    - defaultdict
    - Counter
    - deque
    - ChainMap
    - namedtuple
- The `collections.abc` sub-package provides container Abstract Base Classes, useful for implementing your own containers

Counting Things

- Example: Tabulate total shares of each stock

```
portfolio = [
    ('GOOG', 100, 490.1),
    ('IBM', 50, 91.1),
    ('CAT', 150, 83.44),
    ('IBM', 100, 45.23),
    ('GOOG', 75, 572.45),
    ('AA', 50, 23.15)
]
```

```
{
    ...
    'IBM': 150,
    ...
}
```

- Solution: Use a counter

```
from collections import Counter
total_shares = Counter()
for name, shares, price in portfolio:
    total_shares[name] += shares

>>> total_shares['IBM']
150
```

One-to-Many Mappings

- Problem: Map keys to multiple values

```
portfolio = [
    ('GOOG', 100, 490.1),
    ('IBM', 50, 91.1),
    ('CAT', 150, 83.44),
    ('IBM', 100, 45.23),
    ('GOOG', 75, 572.45),
    ('AA', 50, 23.15)
]
```

```
{
    ...
    'IBM': [(50, 91.1),
        (100, 45.23)],
    ...
}
```

- Solution: use a defaultdict

```
from collections import defaultdict
holdings = defaultdict(list)
for name, shares, price in portfolio:
    holdings[name].append((shares, price))

>>> holdings['IBM']
[(50, 91.1), (100, 45.23)]
```

Keeping a History

- Problem: Keep a history of the last N things

```
line1
line2
line3
line4
line5
...
```

```
history = [line3, line4, line5]
```

- Solution: Use a deque

```python
from collections import deque
history = deque(maxlen=N)
with open(filename) as f:
    for line in f:
        history.append(line)
        ...
```

Exercise 2.5

(10 minutes)

List Comprehensions

- Creates a new list by applying an operation to each element in a sequence

```
>>> a = [1, 2, 3, 4, 5]
>>> b = [2*x for x in a]
>>> b
[2, 4, 6, 8, 10]
```

- Another example:

```
>>> names = ['Elwood', 'Jake']
>>> a = [name.lower() for name in names]
>>> a
['elwood', 'jake']
```

List Comprehensions

- A list comprehension can also filter

```
>>> a = [1, -5, 4, 2, -2, 10]
>>> b = [2*x for x in a if x > 0]
>>> b
[2, 8, 4, 20]
```

- Another example

```
>>> f = open('stockreport', 'r')
>>> goog = [line for line in f if 'GOOG' in line]
```

List Comprehensions

- General syntax

```
[expression for names in sequence if condition]
```

- List comprehensions come from maths

```
a = { x 2 | x ∈ s, x > 0 }     # Math
```

- What it means

```
result = []
for names in sequence:
    if condition:
        result.append(expression)
```

- Can be used anywhere a sequence is expected

```
>>> a = [1, 2, 3, 4]
>>> sum([x*x for x in a])
30
```

List Comp: Examples

- List comprehensions are hugely useful
- Collecting the values of a specific field

```
stocknames = [s['name'] for s in stocks]
```

- Performing database-like queries

```
a = [s for s in stocks if s['price'] > 100
                        and s['shares'] > 50 ]
```

- Data reductions over sequences

```
cost = sum([s['shares']*s['price'] for s in stocks])
```

Exercise 2.6

(15 minutes)

More Details on Objects

- So far: a tour of the most common types
- Have skipped some critical details
- Memory management
- Copying
- Type checking

References and Assignment

- Names (variables) are one way to take a reference to an object
- Python uses reference counting for garbage collection
- There are many ways to take a reference to (store) an object

```
a = value                     # Assignment to a variable
s.append(value)               # Appending to a list
thing.attribute = value       # Setting as an object attribute
d['foo'] = value              # Putting in a dictionary
```

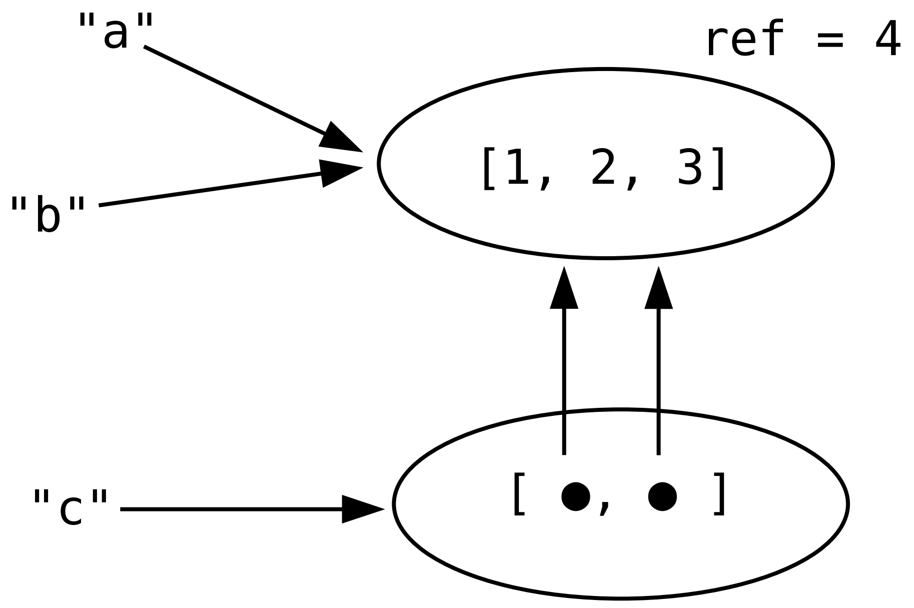
Assignment never copies, it's a reference copy (or pointer copy).

Assignment Example

- Here's some interesting code, how many *different* list objects are there here?

```
>>> a = [1, 2, 3]
>>> b = a
>>> c = [a, b]
```

- Here are the references:



Mutable Objects and References

- Modifying a mutable object by any reference shows up everywhere you have a reference

```
>>> a = [1, 2, 3]
>>> b = a
>>> c = [a, b]
>>> a.append(999)
>>> c
[[1, 2, 3, 999], [1, 2, 3, 999]]
```

This is because no copies were made, all the references point to the same object. This is by design and is not limited to Python.

Call by Object

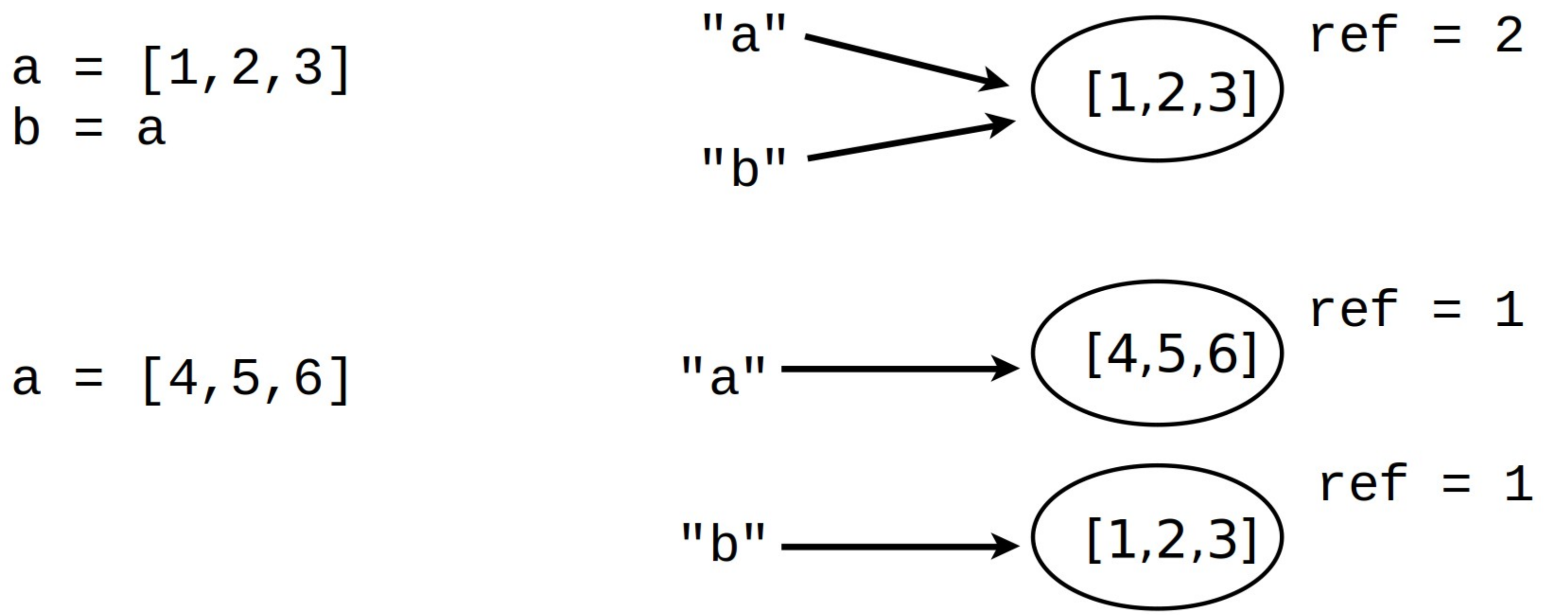
- Functions receive a reference to objects, not a copy

```
>>> def function(thing):
...     thing['new data'] = 33
...
>>> data = {'data': 99}
>>> function(data)
>>> data
{'data': 99, 'new data': 33}
```

- This is useful, not a bug!
- The primitive types (int, float, bool, str) are immutable
- Containers and class instances are *usually* mutable (not tuple or frozenset)

Reassignment

- Reassigning a name (a "rebind" operation) creates a new value rather than modifying the original.



```
a = [1,2,3]
b = a
```

```
a = [4,5,6]
```

- The name "a" points to a new object, "b" is unchanged

Identity versus Equality

- Two objects are equal if they have the same value, but they can be different objects
- We can use the "is" operator to check if two references point to the *same object*

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> c = a
>>> a == b
True
>>> a is c
True
>>> a is not b
True
>>> b.append(999)
>>> a == b
False
```

```
>>> id(a)
140522824988032
>>> id(b)
140522825033216
>>> id(c)
140522824988032
```

Note: id is an integer unique for the lifetime of the object.

Shallow Copies

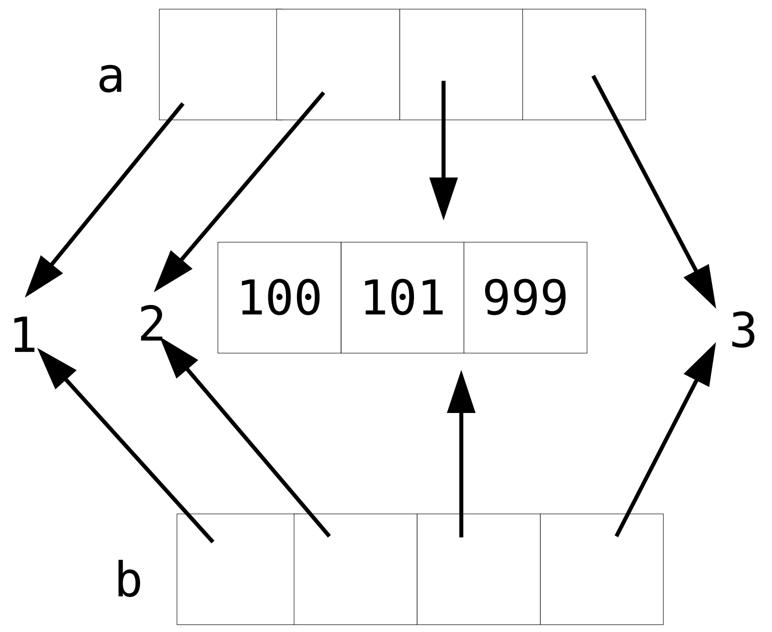
- To avoid problems with mutable objects we can copy them

```
>>> a = [1, 2, [100, 101], 3]
>>> b = list(a)
>>> a is b
False
```

- But notice this:

```
>>> b[2].append(999)
>>> a
[1, 2, [100, 101, 999], 3]
```

- We took a shallow copy, copying references



Deep Copying

- For nested data structures, or objects with shared references, you need to take a "deep copy"
- For this we use the "copy" module

```
>>> import copy
>>> a = [1, 2, [100, 101], 3]
>>> b = copy.deepcopy(a)
>>> b[2].append(999)
>>> a
[1, 2, [100, 101], 3]
>>> a[2] is b[2]
False
```

- There is also copy.copy for shallow copies, but making shallow copies of objects is usually easy

Names, Values, Types

- Names do not have a "type" – it's only a name
- However, values <u>do</u> have an underlying type

```
>>> a = 42
>>> b = 'Hello world'
>>> type(a)
<class 'int'>
>>> type(b)
<class 'str'>
```

- type() will tell you what type it is
- The type will usually be an object you can treat like a function to create or convert a value to that type

Type Checking

- How to tell if an object is a specific type

```
if isinstance(a,list):
    print('a is a list')
```

- Checking for one of several types

```
if isinstance(a, (list,tuple)):
    print('a is a list or tuple')
```

- Don't go overboard with type checking (it adds restrictions, preventing duck typing)
- If possible use the Abstract Base Classes

```
from collections.abc import MutableSequence

if isinstance(a, MutableSequence):
    print('a is a mutable sequence')
```

Everything is an Object

- Everything is an object
- Every object has a type
- No special objects, everything is an object:
  - Numbers and strings
  - Containers
  - Exceptions
  - None and the bools
  - Even classes are objects (so what is the type of a class?)

- All objects can be named and can be passed around as data, placed in containers etc, without restrictions.

In Python we call all objects "first class" objects.

First Class Objects

- A simple example:

```
>>> import math
>>> items = [abs, math, ValueError]
>>> items
[<built-in function abs>, <module 'math' (built-in)>, <class
'ValueError'>]
>>> items[0](-45)
45
>>> items[1].sqrt(2)
1.4142135623730951
>>> try:
...     x = int('not a number')
... except items[2]:
...     print('Failed!')
...
Failed!
```

A list containing a function, a module, and an exception.

You use items in the list in place of the original names.

Summary

- We've looked at the basic principles of working with data in Python programs
- A brief look at part of the object model
- A big part of understanding most Python programs

Exercise 2.7

(15 minutes)

Section 3

Program Organization

Overview

- How to organize larger programs
- Defining and working with functions
- Exceptions and error handling
- Modules
- Script writing

Observation

- A large number of Python programmers spend most of their time writing short "scripts"
- One-off problems, prototyping, data analysis, testing, etc
- Python is good at this!
- And it is what draws many users to Python

What is a "Script"?

- A "script" is a program that runs a series of statements and stops

```
# program.py

statement1
statement2
statement3
...
```

- We've been writing scripts up to this point

Problem

- If you write a useful script it will grow features
- You may apply it to other applications
- You may want to reuse parts in other scripts
- Over time it may become a critical application
- And it might turn into a huge tangled mess
- So let's get organised...

Defining Things

- You must always define things before you use them

```
def square(x):
    return x*x

a = 42
b = a + 2        # Requires that a is already defined

z = square(b)    # Requires square to be defined
```

- The order is important
- Typically variable and function definitions come at the start of the program

Defining Functions

- It is a good idea to put all the code related to a single "task" all in one place

```
def read_prices(filename):
    prices = {}
    with open(filename) as f:
        f_csv = csv.reader(f)
        for row in f_csv:
            name = row[0]
            price = float(row[1])
            prices[name] = price
    return prices
```

- A function simplifies repeated operations (reduces code duplication)
- Well named functions make code easier to read

```
oldprices = read_prices('oldprices.csv')
newprices = read_prices('newprices.csv')
```

What is a Function?

- A function is a sequence of statements

```
def funcname(args):
    statement
    statement
    ...
    return result
```

- Any Python statement can be used inside

```
def foo():
    import math
    print(math.sqrt(2))
    help(math)
```

- There are no "special" statements in Python

Function Definitions

- Functions can be <u>defined</u> in any order

```
def foo(x):
    bar(x)

def bar(x):
    statements
```

```
def bar(x):
    statements

def foo(x):
    bar(x)
```

- Functions must only be defined before they are actually <u>used</u> during program execution

```
foo(3)        # foo must be defined already
```

- Stylistically, it is more common to see functions defined in a "bottom-up" fashion

Bottom-up Style

- Functions are treated as simple building blocks
- The smaller/simpler blocks go first
- The script "entrypoint" is at the bottom

```
# myprogram.py

def foo(x):
    ...
def bar(x):
    ...
    foo(x)
    ...
def spam(x):
    ...
    bar(x)
    ...

spam(42)    # Call spam() to do something
```

Later functions build upon earlier functions.

Code that uses the functions appears at the bottom.

Docstrings

- Documenting the intent of a function in a docstring is good practise

```python
def read_prices(filename):
    '''
    Read prices from a CSV file of name,price
    '''
    prices = {}
    with open(filename) as f:
        f_csv = csv.reader(f)
        for row in f_csv:
            prices[row[0]] = float(row[1])
    return prices
```

- Docstrings are used by help, for tool-tips in IDEs and by documentation generating tools like Sphinx
- And they're helpful when reading code

Type Annotations

- You might see optional type annotations

```
def read_prices(filename: str) -> dict:
    '''
    Read prices from a CSV file of name,price
    '''
    prices = {}
        with open(filename) as f:
            f_csv = csv.reader(f)
            for row in f_csv:
                prices[row[0]] = float(row[1])
    return prices
```

- These are not used at runtime, purely informational
- Used by IDEs and static analysis tools like mypy

Exercise 3.1

(15 minutes)

Default Arguments

- Sometimes you want an optional argument

```
def read_prices(filename, debug=False):
    ...
```

- If a default value is assigned, the argument is optional in function calls

```
d = read_prices('prices.csv')
e = read_prices('prices.dat', True)
```

- Note: arguments with defaults must appear at the end of the argument list (all required arguments go first)

Calling a Function

- Consider a simple function

```
def read_prices(filename, debug):
    ...
```

- Calling with "positional" args

```
prices = read_prices('prices.csv', True)
```

- Calling with "keyword" arguments

```
prices = read_prices(filename='prices.csv',
                     debug=True)
```

- Calling with mixed arguments

```
prices = read_prices('prices.csv', debug=True)
```

Optional/Keyword Arguments

- Arguments with default values are useful for functions that have optional features/flags

```
def parse_data(data, debug=False, ignore_errors=False):
    ...
```

- Compare and contrast calling styles:

```
parse_data(data, False, True)            # ?????

parse_data(data, ignore_errors=True)
parse_data(data, debug=True)
parse_data(data, debug=True, ignore_errors=True)
```

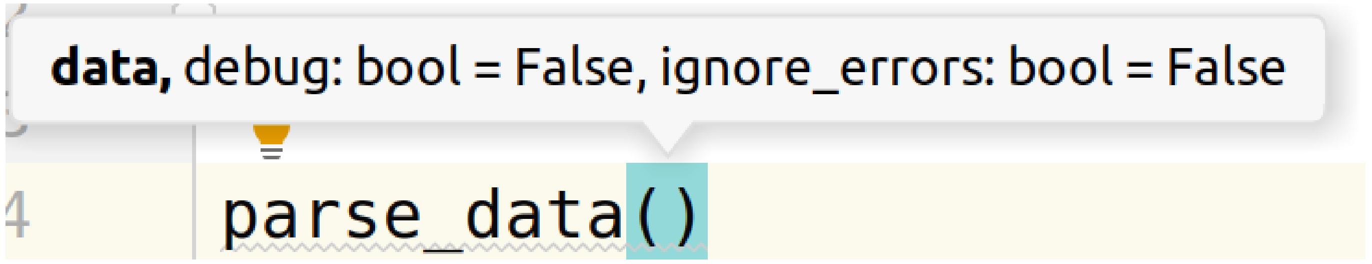
- Keyword arguments improve code clarity
- Optional arguments can be added to functions without breaking existing uses (backwards compatibility)

Design Tip

- Always give short meaningful names to function arguments
- The argument names are part of the API of the function, a design consideration
- Someone using a function may want to use the keyword calling style

```
d = read_prices('prices.csv', debug=True)
```

- Python development tools will show the names in help features and documentation



data, debug: bool = False, ignore_errors: bool = False

```
parse_data()
```

Return Values

- return returns a value

```
def square(x):
    return x*x
```

- return without a value returns None

```
def bar(x):
    statements
    return

a = bar(4)       # a = None
```

- A function without an explicit return, returns None

```
def foo(x):
    statements
    statements

a = foo(9)       # a = None
```

Multiple Return Values

- A function may return multiple values by returning a tuple

```python
def divide(a,b):
    q = a // b        # Quotient
    r = a % b         # Remainder
    return q, r       # Return a tuple
```

- Usage examples:

```python
x, y = divide(37, 5)        # x = 7, y = 2

x = divide(37, 5)           # x = (7, 2)
```

- Unpacking the returned tuple in the call looks like multiple return values

Understanding Variables

- Programs assign values to variables

```
x = value          # Global variable

def foo():
    y = value      # Local variable
```

- Variable assignments occur outside and inside function definitions
- Variables defined outside a function are "global"
- Variables defined inside a function are "local"

Local Variables

- Variables inside functions are private

```
def read_portfolio(filename):
    portfolio = []
    with open(filename) as f:
        for line in f:
            fields = line.split()
            s = (fields[0], int(fields[1]), float(fields[2]))
            portfolio.append(s)
    return portfolio
```

- The names are not available after the function call

```
>>> stocks = read_portfolio('stocks.dat')
>>> fields
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'fields' is not defined
```

- Local variables don't conflict with variables elsewhere

Global Variables

- Functions can access the values of globals

```
name = 'Dave'

def greeting():
    print('Hello', name)
```

- A quirk: functions can't modify globals

```
def spam():
    name = 'Guido'

spam()
print(name)      # prints 'Dave'
```

- <u>All assignments</u> inside a function create local variables

Modifying Globals

- If you must modify a global variable you declare it in the function

```
switch = False

def toggle():
    global switch
    switch = not switch   # Changes the global variable switch
```

- global declaration must occur before use
- Global variables are considered "bad practise" (but common in scripts)
- Avoid globals if you can (use a class instead)

Argument Passing

- When you call a function, the argument variables are names for passed values
- If mutable data types are passed (e.g. lists, dicts), they can be modified "in-place"

```python
def foo(items):
    items.append(42)

a = [1, 2, 3]
foo(a)
print(a)        # [1, 2, 3, 42]
```

- Key point: the function doesn't receive a copy (it gets a new reference to the object)

Understanding Assignment

- Make sure you understand the subtle difference between modifying an object and re-assigning a variable name

- Example:

```
def foo(items):
    items.append(42)    # Modifies items list

def bar(items):
    items = [4,5,6]     # Binds name 'items' to new list
```

- Reminder: variable assignment never overwrites memory (it's a rebind operation, the name is bound to a new value)

Exercise 3.2

(30 minutes)

Error Checking

- Python performs no checking or validation of function argument types or values
- A function will work on any data that is compatible with the statements in the function

- Example:

```
def add(x, y):
    return x + y

add(3, 4)                    # 7
add('Hello', 'World')        # 'HelloWorld'
add('3', '4')                # '34'
```

- This is "duck typing"

Error Checking

- If there are errors in a function they will show up at runtime as exceptions

- Example:

```
def add(x, y):
    return x + y

>>> add(3, '4')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in add
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

- To verify code there is a strong emphasis on testing

Exceptions

- Exceptions are used to signal errors
- Raising an exception (the raise statement)

```
if name not in names:
    raise RuntimeError('Name not found')
```

- Catching an exception (the try...except construct)

```
try:
    authenticate(username)
except RuntimeError as e:
    print(e)
```

Exceptions

- Exceptions propagate up to the first matching except

```python
def foo():
    try:
        bar()
    except RuntimeError as e:
        ...

def bar():
    try:
        spam()
    except RuntimeError as e:
        ...

def spam():
    grok()

def grok():
    ...
    raise RuntimeError('Whoa!')
```

Exceptions

- To handle the exception statements inside the except block run

```
def bar():
    try:
        grok()
    except RuntimeError as e:
        statements
        statements
        ...

def grok():
    ...
    raise RuntimeError('Whoa!')
```

Exceptions

- After handling the exception, execution resumes with the first statement after the try-except

```
def bar():
    try:
        grok()
    except RuntimeError as e:
        statements
        statements
        ...
    statements
    statements
    ...

def grok():
    ...
    raise RuntimeError('Whoa!')
```

Builtin Exceptions

- About two dozen builtin exceptions

```
ArithmeticError
AssertionError
EnvironmentError
EOFError
ImportError
IndexError
KeyboardInterrupt
KeyError
MemoryError
NameError
ReferenceError
RuntimeError
SyntaxError
SystemError
TypeError
ValueError
```

- Consult the Python documentation for more details!

Exception Values

- Most exceptions have an associated value
- More information about what's wrong (the exception message)

```
raise RuntimeError('Invalid user name')
```

- Exception object is supplied to except as a variable

```
try:
    ...
except RuntimeError as e:
    ...
```

- It's an instance of the exception type but can be treated as a string

```
except RuntimeError as e:
    print('Failed : Reason', e)
```

Catching Multiple Errors

- Can catch different kinds of exceptions

```
try:
    ...
except LookupError as e:
    ...
except RuntimeError as e:
    ...
except IOError as e:
    ...
except KeyboardInterrupt as e:
    ...
```

- Alternatively if handling is the same

```
try:
    ...
except (IOError, LookupError, RuntimeError) as e:
    ...
```

Catching All Errors

- Catching any exception

```
try:
    ...
except Exception:
    print('An error occurred')
```

- Overbroad exception handling!
- Don't do this at home

Reraising an Exception

- Use 'raise' to propagate a caught error

```
try:
    go_do_something()
except RuntimeError as e:
    print('Computer says no. Reason :', e)
    raise
```

- Allows you to take action (e.g. logging the error, resource cleanup) but allow the exception to propagate up and be handled at a higher level

Exception Advice

- Don't catch exceptions – fail fast and loud

(if it's important, or possible, to handle the error someone else will handle it)

- Only catch an exception if you're *that* someone
- That is, only catch errors where you can recover and sanely keep going

else statement

- The else statement only runs if no exception occurs
- Useful to minimise the code inside the try-except

- Example:

```
portfolio = []
with open(filename) as f:
    for line in f:
        fields = line.split()
        try:
            s = (fields[0], int(fields[1]), float(fields[2]))
        except ValueError as e:
            print(f'Error parsing line: {e}')
        else:
            portfolio.append(s)
```

- The new value should only be appended if there is no exception

finally statement

- Specifies code that must run regardless of whether or not an exception occurs

```
lock = Lock()
...
lock.acquire()
try:
    ...
finally:
    lock.release()  # release the lock
```

- Commonly used to manage resources that must be cleaned up even if there's an exception (locks, files, etc)

with statement

- In modern code try-finally is often replaced with the 'with' statement

```
lock = Lock()
with lock:
    # lock acquired
    ...
# lock released

with open(filename) as f:
    # Use the file
    ...
# File closed
```

- Define a usage "context" for a resource
- Only works with objects that are "context managers"

Exercise 3.3

(15 minutes)

Modules

- Any Python source file is a module

```
# foo.py
def grok(a):
    ...
def spam(b):
    ...
```

- The import statement loads and executes a module

```
import foo

a = foo.grok(2)
b = foo.spam('Hello')
...
```

- Even the main script/program is run as a module!

Namespaces

- A module is a collection of named values (i.e. it's said to be a "namespace")
- The names are all the global variables and functions defined in the source file
- After import the module name can be used as a prefix

```
>>> import foo
>>> foo.grok(2)
```

- The module name is tied to the source file name (foo → foo.py)

Global Definitions

- Everything defined in the "global" scope is what populates the module namespace

```python
# foo.py
x = 42
def grok(a):
    ...
```

```python
# bar.py
x = 37
def spam(a):
    ...
```

These definitions of x are different

- Different modules can use the same names and those names don't conflict with each other, because each module forms a separate namespace

Modules as Environments

- Modules form an enclosing environment for <u>all</u> of the code defined inside

```python
# foo.py
x = 42

def grok(a):
    print(x)
```

global variables are always bound to the enclosing module (the same file)

- Each source file is it's own little universe
- This is great!
- What happens in a module stays in a module

Module Execution

- When a module is imported, <u>all of the statements in the module execute</u> one after another until the end of the file is reached
- The contents of the module namespace are all of the <u>global</u> names that are still defined at the end of the execution process
- If there are scripting statements that carry out tasks in the global scope (printing, creating files, etc.), you will see them run on import

import as statement

- Changing the name of a module

```
import math as m

def rectangular(r, theta):
    x = r * m.cos(theta)
    y = r * m.sin(theta)
    return x, y
```

- The import happens the same as a normal import
- The only difference is the name of the module afterwards

from module import

- Just imports selected names from a module and makes them available locally

```
from math import sin, cos

def rectangular(r, theta):
    x = r * cos(theta)
    y = r * sin(theta)
    return x, y
```

- Allows parts of a module to be used without having to type the module prefix
- Useful for frequently used features
- Performs slightly better without the attribute lookup (the '.')

Commentary

- Variations on import do <u>not</u> change the way that modules work

```
import math as m
from math import cos, sin
...
```

- import always executes the <u>entire</u> file
- Modules are still an isolated environment
- These variations are only manipulating names

Module Loading

- Each module loads and executes once
- The module is cached on first import
- Repeated imports just return a reference to the previously loaded module
- sys.modules is a dict of all loaded modules

```
>>> import sys
>>> sys.modules.keys()
dict_keys(['sys', 'builtins', '_frozen_importlib', '_imp',
'_warnings', '_io', 'marshal', 'posix', ...]
```

Locating Modules

- When you import Python searches for the module source file
- To find the module Python consults a path list (sys.path)

```
>>> import sys
>>> sys.path
['', '/usr/lib/python310.zip', '/usr/lib/python3.10',
'/usr/lib/python3.10/lib-dynload',
'/usr/local/lib/python3.10/dist-packages',
'/usr/lib/python3/dist-packages']
```

- At the interactive interpreter the current working directory is first
- For a script, the directory containing the script comes first

Module Search Path

- sys.path contains the module search path
- You can manually adjust it if you need (for example to import from a plugin directory or custom module location)

```
import sys
sys.path.append('/project/foo/pyfiles')
```

- You can also use the PYTHONPATH environment variable to add paths

```
$ PYTHONPATH=/project/foo/pyfiles python3.10
Python 3.10.11 (main, Apr  5 2023, 14:15:10) [GCC 9.4.0] on
linux
>>> import sys
>>> sys.path
['', '/project/foo/pyfiles', '/usr/lib/python310.zip',
'/usr/lib/python3.10', ...]
```

Exercise 3.4

(15 minutes)

Main Functions

- In many programming languages there is a concept of a "main" function or method

```
/* C/C++ */
int main(int argc, char *argv[]) {
    ...
}

/* Java */
class myprog {
public static void main(String args[]) {
        ...
    }
}
```

- It's the code that runs when the program is launched
- The entrypoint

Main Module

- Python has no "main" function or method
- Instead, there's a "main" module
- It's the source file that is run first:

```
$ python prog.py
...
```

- Whatever program or script is given at startup is run as the "main" module

__main__ check

- Every module has a name, the `__name__` variable
- The main module is called `"__main__"`
- It's standard practise for modules that *can* run as a main script to use this convention for the entrypoint

```
# prog.py
...
if __name__ == '__main__':
    # Running as the main program
    ...
    statements
    ...
```

- The code inside the __name__ check is the entrypoint for the program

__main__ check

- Important: Any file can run as main *or* be imported
- Consider this simple code:

```
# foo.py
print(__name__)
```

- It behaves differently when run and when imported:

```
$ python foo.py
__main__

>>> import foo
foo
```

- As a general rule you don't want scripting tasks to run when you import code, thus the __main__ check:

```
if __name__ == '__main__':
    # Does not execute if loaded with import
    ...
```

A Program Template

```python
# prog.py

# Import statements (libraries)
import modules

# Functions
def spam():
    ...

def blah():
    ...

# Main function
def main():
    ...

if __name__ == '__main__':
    main()
```

Command Line Tools

- Python is often used for command-line tools

```
$ python report.py portfolio.csv prices.csv
```

- Command line arguments are found in sys.argv

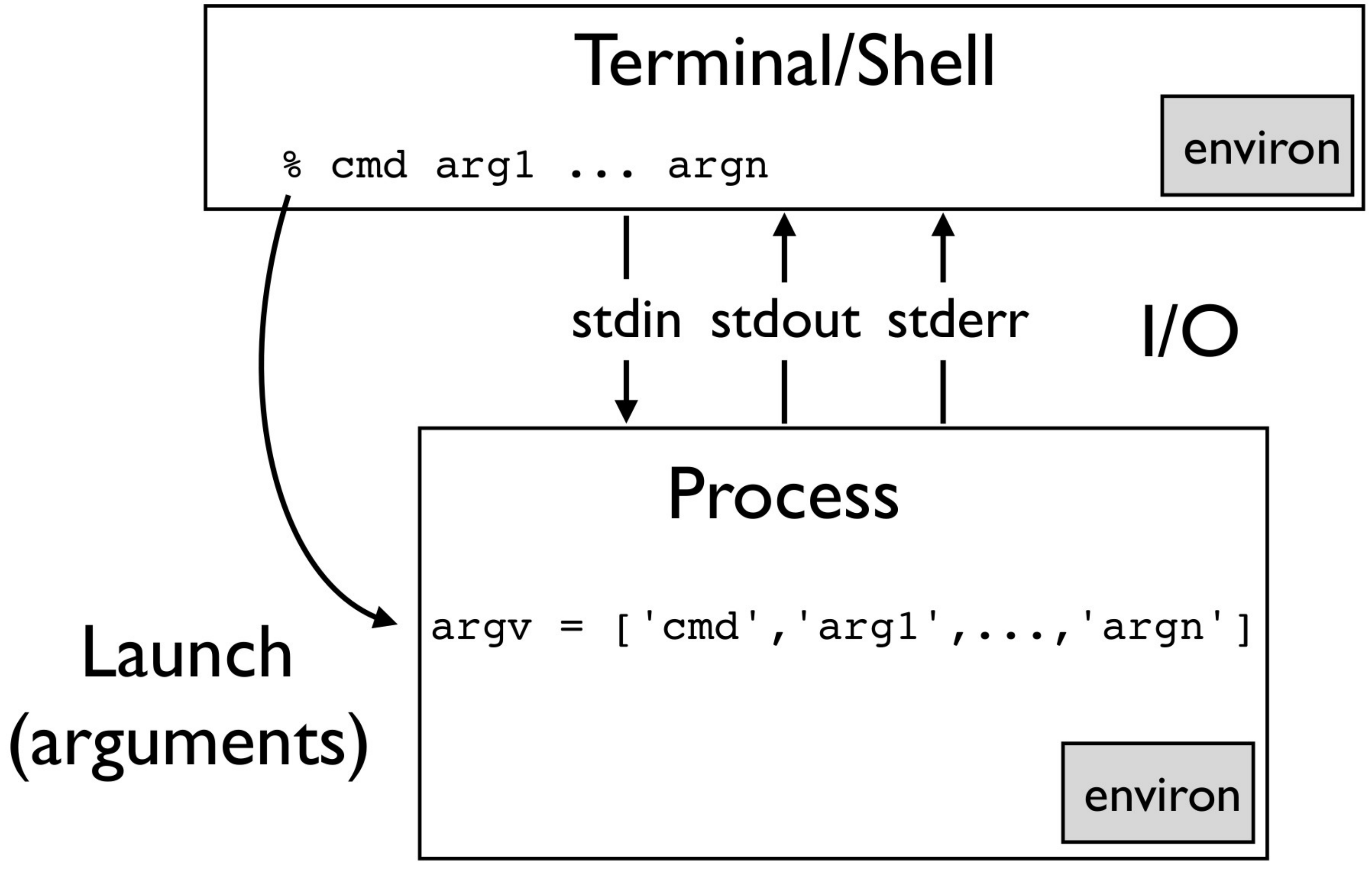
$\texttt{sys.argv} \longrightarrow$ `['report.py', 'portfolio.csv', 'prices.csv']`

- Example of simple processing:

```
import sys
if len(sys.argv) != 3:
     sys.exit(f'Usage: {sys.argv[0]} portfile pricefile')

portfile = sys.argv[1]
pricefile = sys.argv[2]
...
```

- Use argparse in the stdlib for more advanced handling

The Command Line



Command Line Sundries

- Input/output at the shell is done with the `sys.stdin` / `sys.stdout` / `sys.stderr` file handles (streams)
- `print(...)` goes to sys.stdout
- All of the input / output streams might be redirected to or from files at the terminal
- Environment variables can be read and set in `os.environ`, a dictionary like object
- New and changed environment variables are seen by subprocesses
- Program exit is done by raising `SystemExit` or calling `sys.exit(`*`errorcode`*`)`
- A non zero error code on exit is used to indicate error

```
sys.exit(1)
```

Script Template

```
#!/usr/bin/env python
# prog.py

# Import statements (libraries)
import modules

# Functions
def spam():
    ...

def blah():
    ...

# Main function
def main(argv):
# Parse command line args, environment, etc.
    ...

if __name__ == '__main__':
    import sys
    main(sys.argv)
```

Exercise 3.5

(10 minutes)

Design Discussion

provide a filename

```python
def read_data(filename):
    records = []
    with open(filename) as f:
        for line in f:
            ...
            records.append(r)
    return records

d = read_data('Data/file.csv')
```

provide an iterable

```python
def read_data(lines):
    records = []
    for line in lines:
        ...
        records.append(r)
    return records

with open('Data/file.csv') as f:
     d = read_data(f)
```

- Which of these functions do you prefer?
- Why?

Deep Idea: Duck Typing

```
def read_data(lines):
    records = []
    for line in lines:
        ...
        records.append(r)
    return records
```

any "iterable" object
producing strings works

if it walks like a duck and
quacks like a duck we'll
treat it as a duck

```
lines = open('data.csv')

lines = gzip.open('data.csv.gz','rt')

lines = sys.stdin

lines = ['ACME,50,91.1','IBM,75,123.45', ... ]
```

Section 4

Classes and Objects

Object Orientation

- A programming technique where code is organised as a collection of "objects"
- The big idea is that OO wraps up data and functions that operate on it as a single entity
- An "object" consists of:
  - Data (attributes)
  - Methods (functions applied to object)
- You've already been doing it

Object Orientation

- Example: Lists

```
>>> nums = [1, 2, 3]
>>> nums.append(4)              # Method
>>> nums.insert(1, 10)          # Method
```

- nums is an instance of the list type
- methods come from the type, but are attached to the instance
- Think of the methods as functions that take the instance as the first argument:

```
>>> nums = [1, 2, 3]
>>> list.append(nums, 4)
>>> nums
[1, 2, 3, 4]
```

The class statement

- Use 'class' to define a new object

```
class Player:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.health = 100

    def move(self, dx, dy):
        self.dx += dx
        self.dy += dy

    def damage(self, pts):
        self.health -= pts
```

- What is a class?
- Mostly it's a set of functions that carry out operations on so-called "instances"

Instances

- The class definition just creates the class
- Instances are the actual "objects" you use and manipulate in your program
- It's the class (the type) that creates new instances (called "instantiation")
- Created by calling the class like a function:

```
>>> a = Player(2, 3)
>>> b = Player(10, 20)
```

Instance Data

- Each instance has its own local (separate) data

```
>>> a.x
2
>>> b.x
10
```

- The data is initialised by __init__()

```
class Player:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.health = 100
```

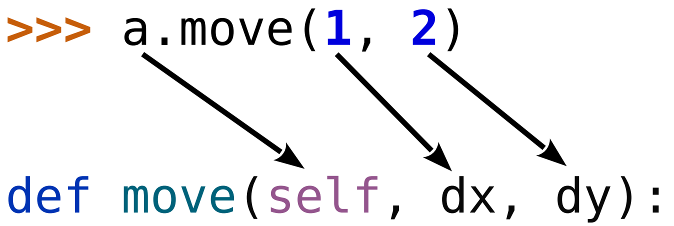
- There are no restrictions on the number or type of attributes stored

Instance Methods

- Functions applied to instances of an object

```python
class Player:
    ...
    def move(self, dx, dy):
        self.x += dx
        self.y += dy
```

- The instance is always passed as the first argument



```python
>>> a.move(1, 2)


def move(self, dx, dy):
```

- By convention the instance is always called "self"
- Basically the same as "this" in other languages

Class Scoping

- Caution: classes do not define a scope

```
??? NameError


class Player:
    ...
    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    def left(self, amt):
        move(-amt, 0)           # NO. Calls global move()
        self.move(-amt, 0)      # YES.
```

- If you want to operate on an instance you <u>always</u> have to refer to it explicitly (using "self")

Exercise 4.1

(15 minutes)

Inheritance

- A tool for specialising objects
- A tool for code reuse!
- Classes without an explicit base inherit from object

```python
class Parent:
    ...

class Child(Parent):
    ...
```

- The parent – also superclass or base class
- The child – also subclass or derived type
- Inheritance can modify and extend the parent

Inheritance

- What do you mean by "specialise"?
- Take an existing class and ...
  - Add new methods
  - Redefine some existing methods
  - Add new attributes

- <u>Reusing and extending existing code</u>

Inheritance Example

- A starting class

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

    def cost(self):
        return self.shares * self.price

    def sell(self, nshares):
        self.shares -= nshares
```

- You can change any part of this via inheritance

Inheritance Example

- Adding a new method

```
class MyStock(Stock):
    def panic(self):
        self.sell(self.shares)
```

```
>>> s = MyStock('GOOG', 100, 490.1)
>>> s.sell(25)
>>> s.shares
75
>>> s.panic()
>>> s.shares
0
```

Inheritance Example

- Redefining an existing method

```
class MyStock(Stock):
    def cost(self):
        return 1.25 * self.shares * self.price

>>> s = MyStock('GOOG', 100, 490.1)
>>> s.cost()
61262.5
```

- Also called "shadowing" or "overriding" a method
- The new method takes the place of the old one
- Other methods are unaffected

Inheritance and Overriding

- Sometimes a class extends an existing method, but it wants to use the original implementation

```
class Stock:
    ...
    def cost(self):
        return self.shares * self.price
    ...

class MyStock(Stock):
    def cost(self):
        actual_cost = super().cost()
        return 1.25 * actual_cost
```

- Use super() to call up to the parent method

Inheritance and __init__

- If __init__ is redefined, you must initialise your parent

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

class MyStock(Stock):
    def __init__(self, name, shares, price, factor):
        super().__init__(name, shares, price)
        self.factor = factor
    def cost(self):
        return self.factor * super().cost()
```

- A particularly common mistake, but easily spotted
- Again, use super()

Using Inheritance

- Sometimes used to organize related objects

```python
class Shape:
    ...
class Circle(Shape):
    ...
class Rectangle(Shape):
    ...
```

- Think logical hierarchy or taxonomy
- Example: the exception type system

Using Inheritance

- More commonly used as a code-reuse tool

```python
class CustomHandler(TCPHandler):
    def handle_request(self):
        ...
        # Custom processing
```

- Base class contains general purpose code
- You inherit to customise specific parts
- Maybe it plugs into a framework

"is a" relationship

- Inheritance establishes an "is a" relationship

```
class Shape:
    ...
class Circle(Shape):
    ...

>>> c = Circle(4.0)
>>> isinstance(c, Shape)
True
>>> isinstance(c, object)
True
```

- Important: Code that works with the parent is also supposed to work with the child
- This is the "Liskov Substitution Principle"

Multiple Inheritance

- You can specify multiple base classes

```
class Mother:
    ...
class Father:
    ...
class Child(Mother, Father):
    ...
```

- The new class inherits from both the parents
- But there are some rather tricky details
- Don't do this unless you know what you're doing!
- Only use multiple inheritance for mixin classes

Exercise 4.2

(30 minutes)

Special Methods

- Classes may define special methods
- Known as: protocol / dunder / magic methods
- They are normally called by Python rather than directly by the user
- They have special meaning to the Python interpreter
- Always proceeded and followed by double-underscore

```
class Stock:
    def __init__(self):
        ...
    def __repr__(self):
        ...
```

- There are dozens!
- We'll look at a few examples

String Representation

- Objects can have two different string representations

```
>>> from datetime import date
>>> d = date(2023, 5, 14)
>>> print(d)
2012-05-14
>>> d
datetime.date(2023, 5, 14)
```

- str(x) – printable output

```
>>> str(d)
'2023-05-14'
```

- repr(x) – for programmers

```
>>> repr(d)
'datetime.date(2023, 5, 14)'
```

String Conversions

```
class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day
    def __str__(self):
        return f'{self.year}-{self.month}-{self.day}'
    def __repr__(self):
        return f'Date({self.year},{self.month},{self.day})'
```

Note: the convention for __repr__() is to return a string that if fed to eval will recreate the object. This isn't always possible, in which case return some useful representation.

Note: Instead of hardcoding the class you can use `f'{self.__class__.__name__}'` which works better with subclasses.

The Numeric Protocol

- Mathematical operations (see reference for the gory details including right hand variants)

`a + b`	`a.__add__(b)`	Addition
`a - b`	`a.__sub__(b)`	Subtraction
`a * b`	`a.__mul__(b)`	Multiplication
`a / b`	`a.__div__(b)`	Division
`a // b`	`a.__floordiv__(b)`	Floor division
`a % b`	`a.__mod__(b)`	Modulo
`a << b`	`a.__lshift__(b)`	Left shift
`a >> b`	`a.__rshift__(b)`	Right shift
`a & b`	`a.__and__(b)`	Bitwise and
`a | b`	`a.__or__(b)`	Bitwise or
`a ^ b`	`a.__xor__(b)`	Bitwise xor
`a ** b`	`a.__pow__(b)`	Power
`-a`	`a.__neg__()`	Unary negative
`~a`	`a.__invert__()`	Bitwise not
`abs(a)`	`a.__abs__()`	Absolute value
`a @ b`	`a.__matmul__(b)`	Matrix multiplication

The Container Protocol

- Methods for implementing containers

```
len(x)          x.__len__()
x[a]            x.__getitem__(a)
x[a] = v        x.__setitem__(a)
del x[a]        x.__delitem__(a)
a in x          x.__contains(a)
```

- Definitions in a class

```
class Container:
    def __len__(self):
        ...
    def __getitem__(self, a):
        ...
    def __setitem__(self, a, v):
        ...
    def __delitem__(self, a):
        ...
    def __contains__(self, a):
        ...
```

Odds and Ends

- Defining new exceptions
- Bound and unbound methods
- Attribute lookups

Defining Exceptions

- User defined exceptions are defined as classes

```
class NetworkError(Exception):
    pass
```

- Exceptions always inherit from Exception
- Usually it's an empty class, no extra work to do
- You can also make a hierarchy

```
class AuthenticationError(NetworkError):
    pass

class ProtocolError(NetworkError):
    pass
```

Method Invocation

- Invoking a method is a two-step process
- Lookup: the . Operator
- Method call: the () operator

```
class Stock:
    ...
    def cost(self):
        return self.shares * self.price

>>> s = Stock('GOOG', 100, 490.10)
>>> c = s.cost
>>> c
<bound method Stock.cost of <__main__.Stock object at
0x7f2f200de1d0>>
>>> c()
49010.0
```

$$\texttt{c = s.cost} \xleftarrow{} \text{Lookup}$$

$$\texttt{c()} \xleftarrow{} \text{Method call}$$

Bound Methods

- A method that has not yet been invoked by the function call operator () is known as the "bound method"
- The first argument, self, is already bound in
- The bound method operates on the instance it came from

```
>>> s = Stock('GOOG', 100, 490.10)
>>> s
<__main__.Stock object at 0xde1d0>
>>> c = s.cost
>>> c
<bound method Stock.cost of <__main__.Stock object at 0xde1d0>>
>>> c()
49010.0
```

binding

Attribute Access

- There are four builtin functions for accessing attributes with a string variable

```
getattr(obj, 'name')                # Same as obj.name
setattr(obj, 'name', value)         # Same as obj.name = value
delattr(obj, 'name')                # Same as del obj.name
hasattr(obj, 'name')                # Tests if attribute exists
```

- Example: probing for an optional attribute

```
if hasattr(obj, 'seek'):
    obj.seek(0)
```

- Note: getattr has a useful default value argument

```
x = getattr(obj, 'x', None)
```

Exercise 4.3

(15 minutes)

namedtuples

- For simple data classes (that only hold data) you might consider namedtuples instead
- They are a subclass of tuple so you can replace functions/methods that return tuples with namedtuples instead – and remain backwards compatible
- They provide attribute access and a nice string representation
- namedtuple is a class factory, it builds classes, and it lives in the collections module

namedtuples

```
>>> from collections import namedtuple
>>> Stock = namedtuple('Stock', ['name', 'shares', 'price'])
>>> Stock
<class '__main__.Stock'>

>>> s = Stock('GOOG', 100, 490.1)
>>> s
Stock(name='GOOG', shares=100, price=490.1)
>>> s.name
'GOOG'
>>> s.shares
100
>>> s.price
490.1
>>> name, shares, price = s
>>> name, shares, price
('GOOG', 100, 490.1)
```

dataclasses

- New in Python 3.7
- The trouble with returning tuples is that adding an extra return value is then incompatible with existing code
- Returning an object that isn't a tuple is more flexible (more attributes can be added to the return value)
- Use namedtuples for code that already returns tuples, use dataclasses for new code
- You use dataclass as a "class decorator" to make the class and use type annotations to declare members
- dataclasses make it easier to add methods as well

dataclasses

```
from dataclasses import dataclass

@dataclass
class Stock:
    name: str
    shares: int
    price: float

    def cost(self):
        return self.shares * self.price

>>> s = Stock('GOOG', 100, 490.10)
>>> s
Stock(name='GOOG', shares=100, price=490.1)
>>> s.name
'GOOG'
>>> s.shares
100
>>> s.price
490.1
>>> s.cost()
49010.0
```

Section 5

Encapsulation and Properties

Encapsulation

- Encapsulation is one of the four pillars of Object Orientation:
  - Abstraction
  - Inheritance
  - Polymorphism
  - Encapsulation
- Objects encapsulate data and the private implementation details of the object
- There's a distinction between the "public API" and the private implementation details (which may change)
- This is design thinking – how should your objects be used from the outside, and which parts are you free to change

The Problem with Python

- Python still makes an important distinction between public and private
- But in Python everything is "open by default"
  - All methods and attributes are visible
  - You can access, call and modify just about everything
  - There's no concept of truly "private members"
- Python provides encapsulation but it's a "translucent encapsulation" (you can see into it)
- If you want to mark parts of your code as private implementation details it seems like this is a problem

Note: even languages that have private data provide reflection (or you can use pointers) to circumvent this.

Python and Encapsulation

- Python provides encapsulation, private methods and attributes, through a programming convention
- It's based on naming
- Python is based on the principle of "consenting adults", if there are things you shouldn't do the right answer is "don't do them"
- But having things open by default can be really useful for testing and those times where you really have to access private data

Private Attributes

- Any attribute (or method) whose name starts with a leading underscore (_) is considered private

```
class Person:
    def __init__(self, name):
        self._name = name
```

- However it's only a programming convention
- The attributes can still be accessed
- But everyone understands this convention

```
>>> p = Person('Michael')
>>> p._name
'Michael'
```

- If you mess with private attributes you get what you deserve!

Problem: Simple Attributes

- Consider our simple Stock class

```
class Stock(object):
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

s = Stock('GOOG', 100, 490.1)
s.shares = 50
```

- There's nothing to prevent a user accidentally setting an attribute to the wrong type

```
s.shares = '50'     # --> TypeError
```

- How can we add validation to prevent this?

Managed Attributes

- Accessor methods (getters and setters) are one possibility

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.set_shares(shares)
        self.price = price

    def get_shares(self):
        return self._shares

    def set_shares(self, value):
        if not isinstance(value, int):
            raise TypeError('Expected an int')
        self._shares = value
```

methods that layer get/set operations on top of a private attribute

- Too bad this breaks all existing code

$$\texttt{s.shares = 50} \longrightarrow \texttt{s.set\_shares(50)}$$

Properties

- An alternative approach to accessor methods

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.set_shares(shares)
        self.price = price

    @property
    def shares(self):
        return self._shares

    @shares.setter
    def shares(self, value):
        if not isinstance(value, int):
            raise TypeError('Expected an int')
        self._shares = value
```

getter method

setter method

- The syntax can appear a little weird at first

Properties

- Normal attribute access triggers the property
- Fully compatible with existing code

```python
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.set_shares(shares)
        self.price = price

    @property
    def shares(self):
        return self._shares

    @shares.setter
    def shares(self, value):
        if not isinstance(value, int):
            raise TypeError('Expected an int')
        self._shares = value
```

```
>>> s = Stock(...)
>>> s.shares
100
>>> s.shares = 50
```

Properties

- You don't change existing code
- Attribute access triggers the property (via the "descriptor protocol")

```
class Stock:
    def __init__(self, name, shares, price):
        ...
        self.shares = shares
        ...

    @shares.setter
    def shares(self, value):
        if not isinstance(value, int):
            raise TypeError('Expected an int')
        self._shares = value
```

assignment calls the setter

- The public api is the "shares" property, the data is stored in the private attribute "_shares"

Properties

- Also useful for computed data attributes
- "Get only" properties (no setter) are valid

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

    @property
    def cost(self):
        return self.shares * self.price
```

- Example:

```
>>> s = Stock('GOOG', 100, 490.1)
>>> s.cost    <------------------------  computed attribute
49010.0
```

Exercise 5.1

(10 minutes)

Section 6

Additional Language Features

More Python

- A few very useful bits of Python syntax we haven't already covered
- Some of them from more recent versions of Python
    - Generator expressions
    - Ordered dictionaries
    - Ternary expressions
    - The walrus operator
    - Positional and keyword only arguments

Generator Expressions

- List comprehensions are "eager", they consume their input and produce a list
- Many of the builtin functions in Python are "lazy", they produce iterable objects instead of executing immediately

```
>>> range(100)
range(0, 100)
>>> zip(['name', 'shares', 'prices'], ['GOOG', 100, 490.10])
<zip object at 0x7f503b5d80c0>
>>> enumerate(nums)
<enumerate object at 0x7f503b6a8840>
```

- Generator expressions are a lazy version of list comprehensions

Generator Expressions

- Generator expressions produce "one shot" generators
- The syntax is very similar to list comprehensions

```
>>> a = [2, 4, 6, 8, 10]
>>> b = (x**2 for x in a)
>>> b
<generator object <genexpr> at 0x7f503b78f760>
>>> for result in b:
...     print(result)
...
4
16
36
64
100
```

- They don't produce a list, so the whole result set doesn't need to be in memory
- They can't be reused

Generator Expressions

- General syntax (very similar to list comprehensions)

```
(expression for names in iterable if conditional)
```

- They look better than list comprehensions in function calls

```
sum(x*x for x in a)
```

- Can be applied to any iterable and even chained together

```
>>> a = [1,2,3,4]
>>> b = (x*x for x in a)
>>> c = (-x for x in b)
>>> for i in c:
...     print(i, end=' ')
...
-1 -4 -9 -16
```

Ordered Dictionaries

- Since Python 3.7 Python dictionaries are now ordered by insertion order
- Originally a memory (layout) optimisation originating in pypy and ported to CPython
- Iteration over dictionaries, the keys and values and items, all preserve order

```
>>> d = {}
>>> d['first'] = 1
>>> d['second'] = 2
>>> d['third'] = 3
>>> list(d)
['first', 'second', 'third']
>>> d.keys()
dict_keys(['first', 'second', 'third'])
>>> d.items()
dict_items([('first', 1), ('second', 2), ('third', 3)])
>>> d.values()
dict_values([1, 2, 3])
```

Ternary Expressions

- Also known as "conditional expressions"
- A concise way of having an expression evaluate to a value based on a condition

```
>>> email_address = None
>>> send_email = True if email_address is not None else False
>>> send_email
False
>>> email_address = 'michael@python.org'
>>> send_email = True if email_address is not None else False
>>> send_email
True
```

- Very terse syntax, it maybe clearer to use if/else
- Like list comprehensions it can be helpful to start reading them in the middle (the true condition is on the left, the false is on the right and the if is in the middle)

The Walrus Operator

- Assignment as an expression: `x := 3`
- New in Python 3.8
- Useful where you need to test a value immediately after setting it
- Can be used to write inscrutable code!

- Old code (regular expressions):

```
>>> match = re.match(r'\w+@(\w+\.\w+)', email_address)
>>> if match is not None:
...     domain = match.groups(1)
...
```

- With the walrus operator:

```
>>> if match := re.match(r'\w+@(\w+\.\w+)', email_address):
...     domain = match.groups(1)
...
```

Positional and Keyword Only Arguments

- Python function signatures are now very rich
- We can now express positional and keyword only arguments
- Positional only arguments (mostly for compatibility with C functions) added in Python 3.8
- Keyword only arguments were new in Python 3.0

```
>>> def foo(data, /, *, debug=False):
...     pass
...
>>> foo(1, debug=True)
>>> foo(data=2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got some positional-only arguments passed as
keyword arguments: 'data'
>>> foo(3, False)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() takes 1 positional argument but 2 were given
```