MONASH University

School of Physics and Astronomy
ASP4000
2015

APS4000 Introduction to Honours Computing

Introduction to Python

Part I

Basics

1 Introduction

Python is a object-oriented interpreted script language. There is also a recent nature article at `http://www.nature.com/news/programming-pick-up-python-1.16833`. For documentation and tutorial, I generally recommend the original online python documentation at `https://docs.python.org/3/`. Of course, you can always "google" for specific recipes. Python allows you to work interactively, write routines you can call, but also to write standalone applications. There is modules for almost anything. Python is used for applications like running web servers to super computing. It also allows to interface to other languages, e.g., FORTRAN (f2py), C/C++, R, ... You can use python for scripted text processing, data analysis (numpy), plotting/visualization (matplotlib). The later two packages we will use in the second part of this course. Here we will first focus on a basic introduction to python3. This introduction is not comprising, it just is meant to give you an idea of the power of python, what you all might be able to do, but may have to look up later. It will not replace you actually reading the manual and tutorial, which I highly recommend.

Python is object-oriented – in Python *everything* is an object. ***Everything.*** Object oriented programming (`http://en.wikipedia.org/wiki/Object-oriented_programming`) combines data and code, allowing you data encapsulation and includes inheritance and polymorphism.

In this crash course - and this will crash any person's ability to absorb it all in just 3h - I will give an overview so that you have seen what you may be able to all use in python, to inspire you, to model you research projects and ideas in python, and find the right data structures and organization for it. So that later you may remember having seen things that you could use. I do not expect that you remember all by heart. I need to look up and try out things continuously myself.

2 Starting Python

In this course we want to use Python 3. At the time of this writing, the current version is Python 3.4.3. For simpler editing, we use IPython. The current version is 2.4.1.

In the following the shell prompt (Bash) is displayed as

```
~>
```

To start IPython we use

```
~> ipython
```

you should then see a message like

```
Python 3.4.2 (default, Oct 14 2014, 06:44:47)
Type "copyright", "credits" or "license" for more information.

IPython 2.4.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

Now you are ready to use python. There is also more fancy shells, e.g., the IPython notebook, which you would start using

```
~> ipython notebook
```

This should pop up a new tab in your web browser. [1] The you press the `New Notebook` button on the top right.

Very useful later in this course - and indispensable for you later when developing python codes - will be to automatically reload modules when you modify them.

You can do the manually, every time you start IPython, by first loading the `autoreload` extension, then activating it to reload all loaded modules automatically when changed.

```
In [2]: %load_ext autoreload

In [3]: %autoreload 2
```

Instructions on how to set this up by default so it is done automatically every time you start IPython can be found at, e.g., `https://www.reddit.com/r/Python/comments/rsfsi/tutorial_spend_30_seconds_setting_up/`

[1] For me this required installation of `pyzmq` and `jinja2`.

3 Basic Data Types

3.1 Scalar data types

3.1.1 Integer

Just the number. Can be arbitrary large. Must not contain ".". Example

```
In [4]: 2
Out[4]: 2

In [5]: 200000000000000000000000000000000000000000000000000000000000000000000000
Out[5]: 200000000000000000000000000000000000000000000000000000000000000000000000
```

Integer constants can also be specified in other bases, e.g.,

```
In [6]: 0x32
Out[6]: 50

In [7]: 0o32
Out[7]: 26

In [8]: 0b101
Out[8]: 5
```

3.1.2 Floating Point

Floating point number with finite precision. Use "." to separate fraction part. Use Just the number. Use "e" to separate exponent. Example:

```
In [9]: 2.
Out[9]: 2.0

In [10]: -2.e23
Out[10]: -2e+23

In [11]: 2.0001e23
Out[11]: 2.0001e+23

In [12]: 2.00000000000000000000000000000001e23
Out[12]: 2.0000000000000002e+23
```

Precision is system-dependent, typical is IEEE-754 (`http://en.wikipedia.org/wiki/IEEE_floating_point` 8-byte binary floating point (15 digits precision, exponent ± 300. Classes for (internal) decimal representation exists as well.

3.1.3 Complex Numbers

Use "j" for imaginary part.

```
In [13]: 3 -4j
Out[13]: (3-4j)
```

3.1.4 Operators

The usual, `+`, `-`, `*`, `/`, `%` (modulo), `**` (power). Special: `//` (integer division). Combining integer with float will result in float. Examples:

```
In [14]: 7 // 3
Out[14]: 2

In [15]: 7 / 3
Out[15]: 2.3333333333333335

In [16]: 7. // 3
Out[16]: 2.0

In [17]: 10**400 + 1.
---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)
<ipython-input-22-c0af5dd83786> in <module>()
----> 1 10**400 + 1.

OverflowError: int too large to convert to float
```

There is also bit-wise binary operations on integers using `&` (and), `|` (or), `^` (xor), `~` (not), "<<" (left shift), >> (right shift)

```
In [18]: 7 & 3
Out[18]: 3

In [19]: 7 | 8
Out[19]: 15

In [20]: 7 ^ 3
Out[20]: 4

In [21]: ~3
Out[21]: -4

In [22]: 3 << 2
Out[22]: 12
```

3.1.5 Truth and Logical Operations

Define logical values “True” and “False”. Logical operations include `or`, `not`, and `and`:

```
In [23]: True or False
Out[23]: True

In [24]: True and False
Out[24]: False

In [25]: not True
Out[25]: False
```

3.1.6 Comparison Operators

These include <, >, <=, == (equality, in contrast to assignment), ! = (not equal), `is` (object identity), and `is not` (negated object identity).

```
In [26]: (1 > 3) or (3 == 4)
Out[26]: False
```

3.1.7 Nothing

... and there is the `None` object - we will use later

```
In [27]: None
```

... and the `Ellipsis` object

```
In [28]: Ellipsis
Out[28]: Ellipsis
```

3.1.8 Strings

Sequence of characters enclosed by batching single or double quotation marks. Multi-line strings can be defined using triple quotation marks.

```
In [29]: 'abc 123'
Out[29]: 'abc 123'

In [30]: """abc
   ....: 123"""
Out[30]: 'abc\n123'
```

Here, special characters line `new line` start with a
. You can add them manually:

```
In [31]: 'abc\n123'
Out[31]: 'abc\n123'
```

To input special characters in to a string w/o interpretation, use a *raw string* with has an `r` in front of the string:

```
In [32]: r'abc\n123'
Out[32]: 'abc\\n123'
```

Here the backslash itself is represented by a double backslash. To see the difference, we can use the `print` function

```
In [33]: print('abc\n123')
abc
123

In [34]: print(r'abc\n123')
abc\n123
```

You can also add strings or replicate strings

```
In [35]: 'abc' + '123'
Out[35]: 'abc123'

In [36]: '12' * 12
Out[36]: '121212121212121212121212'
```

There is a whole variety of other string methods to be discussed later or in the manual. Note that in Python 3 there is also a string-like data types `bytes`, `bytearray`, and `memoryview` but these are for more advanced use cases - and to keep competent python programmers employed.

```
In [37]: bytearray(12)
Out[37]: bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')

In [38]: b'1234'
Out[38]: b'1234'
```

3.1.9 Indexing and Slicing

Slicing and indexing generally can be applied to all ordered data that can be indexed. It is done with the "[_]" operator, where "_" stands for an argument.

For strings, you can get individual characters (index is base-0)

```
In [39]: 'abc123'[2]
Out[39]: 'c'
```

or sub-strings (called "slicing" - **last index excluded!**). The basic syntax is "start:stop[:step]". Default step size is 1 and omitted start/stop values run to the end of the structure (string). Negative values count from the back, with -1 referring to the last element.

```
In [40]: 'abc123'[2]
Out[40]: 'c'

In [41]: 'abc123'[2:4]
Out[41]: 'c1'

In [42]: 'abc123'[2::2]
Out[42]: 'c2'

In [43]: 'abc123'[::-2]
Out[43]: '31b'

In [44]: 'abc123'[:-1]
Out[44]: 'abc12'

In [45]: 'abc123'[-2:0:-1]
Out[45]: '21cb'
```

This is the default behaviour of slicing, but in principle, each object can define how it wants to react to this, so can objects (classes) you define by defining the necessary attributes. More later.

3.2 Organizing Data - Variables

You can also assign values to variables.

```
In [46]: a = 12

In [47]: print(a)
12
```

Variable can have letters, undershorts and numbers but must not start with a number. Variable names starting with one or two underscores usually have special meaning. Variable names are case sensitive.

Note: **A variable is a *name* (pointer to) that object. Assignment to an existing variable does change where it points to not the object that it points to.**

```
In [48]: a = 12

In [49]: b = a

In [50]: a = 13
```

```
In [51]: b
Out[51]: 12
```

Variables do not need to be defined and given a type in advance; the type comes with the object it points to.

```
In [52]: a = 12

In [53]: a = 'abc'

In [54]: a
Out[54]: 'abc'
```

In-place assignment operators are short hand for written-out expression[2] This way you can *apparently* add even to non-mutable objects like strings. Operators comprise the usual suspects, `+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `&=`, `|=`, `^=`, ...

```
In [55]: i = 3

In [56]: i += 4.

In [57]: i
Out[57]: 7.0

In [58]: s = 'abc'

In [59]: s += 'd'

In [60]: s
Out[60]: 'abcd'
```

In the string case, "`s`" is instead now pointing to a new string object. Later we will see for `numpy` that mutable objects can modify this behavior.

3.3 Organizing Data - Containers

3.3.1 Lists

List of elements in square brackets

```
In [61]: [1, 2, 3]
Out[61]: [1, 2, 3]
```

[2]except that they call, e.g., for addition, the object's `__iadd__` method rather than `__add__`/`__radd__` methods.

Adding lists and replicating list elements

```
In [62]: [1, 2, 3] + [4, 5, 6]
Out[62]: [1, 2, 3, 4, 5, 6]

In [63]: [1,2,3] * 4
Out[63]: [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]

In [64]: a = [1, 2, 3]

In [65]: a += [4]

In [66]: a
Out[66]: [1, 2, 3, 4]

In [67]: a += [4, 5]

In [68]: a
Out[68]: [1, 2, 3, 4, 4, 5]
```

Assigning to list elements

```
In [69]: a = [1, 2, 3]

In [70]: a[1]=4

In [71]: a
Out[71]: [1, 4, 3]

In [72]: a[1]='c'

In [73]: a
Out[73]: [1, 'c', 3]

In [74]: a[0]=a

In [75]: a
Out[75]: [[...], 'c', 3]

In [76]: a[0][0][0][1][0]
Out[76]: 'c'
```

That is, list entries can be any kind of object, even itself, and lists are *mutable*. In contrast, strings are not mutable.

```
In [77]: s = '123'
```

```
In [78]: s[2] = 'a'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-68-fc108b9c3521> in <module>()
----> 1 s[2] = 'a'

TypeError: 'str' object does not support item assignment
```

You can even replace ranges by ranges

```
In [79]: a = [1, 2, 3]

In [80]: a[0:2] = [4, 5, 6]

In [81]: a
Out[81]: [4, 5, 6, 3]
```

But note that assignment to elements is different

```
In [82]: a = [1, 2, 3]

In [83]: a[1] = [1, 2, 3]

In [84]: a
Out[84]: [1, [1, 2, 3], 3]
```

and that ranges cannot be replaced by scalars

```
In [85]: a[1:2] = 1
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-79-6e71711e9dd0> in <module>()
----> 1 a[1:2] = 1

TypeError: can only assign an iterable
In [86]: a[1:2] = [1]

In [87]: a
Out[87]: [1, 1, 3]
```

There is also specific list functions and methods, e.g., `append`, `len`, `index`, `count`, `min`, `max`, `copy`, `insert`, `clear`, `remove`, `pop`, `reverse`, `sort`, ... and the `sorted` function

```
In [88]: a = [1, 3, 2]

In [89]: a.sort()
```

```
In [90]: print(a)
[1, 2, 3]

In [91]: sorted([3, 5, 2, 4])
Out[91]: [2, 3, 4, 5]
```

In the example above, the "()" stand for a function call, here w/o any parameter.

Empty lists:

```
In [92]: []
Out[92]: []

In [93]: list()
Out[93]: []
```

You can test whether an element is in the list

```
In [94]: 2 in [1, 2, 3]
Out[94]: True

In [95]: 2 not in [1, 2, 3]
Out[95]: False
```

Number of elements

```
In [96]: len([1, 2, 3])
Out[96]: 3
```

Deleting elements

```
In [97]: a = [1, 2, 3]

In [98]: del a[1]

In [99]: a
Out[99]: [1, 3]
```

3.3.2 Tuples

Like lists, but not mutable. Generated by the comma operator, enclosed by bracket if ambiguous otherwise.

```
In [100]: a = (1, 'a', [1])

In [101]: a[2]
Out[101]: [1]
```

Empty and 1-element tuples:

```
In [102]: a = ()

In [103]: a
Out[103]: ()

In [104]: a = (1,)

In [105]: a
Out[105]: (1,)
```

Except operations that *change* the tuple, most list operations work for tuples as well.

```
In [106]: (1,) * 4
Out[106]: (1, 1, 1, 1)

In [107]: 1 in (1, 2, 3)
Out[107]: True
```

3.3.3 Dictionaries

Dictionaries are very efficient to organize data that cannot be indexes easily. It consists of a pair of key and index; the key has to be a "hashable" (i.e., usually non-mutable) object like a number, string, or Tuple. Truth values, None, and Ellipsis are fine as well.) Dictionaries are nor ordered or sorted. Create with square brackets or `dict` function.

```
In [108]: a = {'a': 3, True: 4, 5: 7, (1,2): 4}

In [109]: a
Out[109]: {(1, 2): 4, True: 4, 5: 7, 'a': 3}

In [110]: b = dict(a=7, c=11)

In [111]: b
Out[111]: {'c': 11, 'a': 7}
```

Note that in the last example the keywords are converted to strings.

You can test whether a key is present

```
In [112]: 'c' in b
Out[112]: True
```

You can combine dictionaries using `update`

```
In [113]: a.update(b)

In [114]: a
Out[114]: {(1, 2): 4, True: 4, 'c': 11, 5: 7, 'a': 7}
```

and access elements using the indexing syntax or the get *method*

```
In [115]: a[True]
Out[115]: 4

In [116]: a.get(True)
Out[116]: 4
```

Get a default value if key is not defined

```
In [117]: a.get(False, 0)
Out[117]: 0
```

remove elements (element returned)

```
In [118]: a.pop(True)
Out[118]: 4

In [119]: a
Out[119]: {(1, 2): 4, 'c': 11, 5: 7, 'a': 7}
```

set default values for undefined entries

```
In [120]: a.setdefault(False, 0)
Out[120]: 0

In [121]: a
Out[121]: {(1, 2): 4, False: 0, 'c': 11, 5: 7, 'a': 7}
```

New values can be easily added, by assignment; if the key does not yet exists it is added, if it dies exist, it is overwritten.

```
In [122]: a = {}

In [123]: a['a'] = 1

In [124]: a['b'] = 2

In [125]: a
Out[125]: {'b': 2, 'a': 1}
```

```
In [126]: a['a'] = 3

In [127]: a
Out[127]: {'b': 2, 'a': 3}
```

Number of elements

```
In [128]: len(a)
Out[128]: 2
```

Deleting elements

```
In [129]: a = dict(a=2,b=3)

In [130]: a
Out[130]: {'b': 3, 'a': 2}

In [131]: del a['b']

In [132]: a
Out[132]: {'a': 2}
```

There are variation classes like OrderedDict as well.

3.3.4 Sets

Sort of like dictionaries, but only keys, no values.

```
In [133]: {1, 2, 3}
Out[133]: {1, 2, 3}

In [134]: {1, 2, 3} | {3, 4, 5}
Out[134]: {1, 2, 3, 4, 5}

In [135]: set()
Out[135]: set()

In [136]: {1, 'a', (1, 2)}
Out[136]: {(1, 2), 1, 'a'}
```

Set operations include <= (subset), < (proper subset), >= (superset), > (proper superset), | (union), `&` (intersection), − (difference), ^ (symmetric difference), ...

There is also a hashable (non-mutable) set variation, frozenset.

3.3.5 Converting Between Types

Dictionary to list

```
In [137]: b = dict(a = 7, c = 11)

In [138]: list(b)
Out[138]: ['c', 'a']

In [139]: list(b.items())
Out[139]: [('c', 11), ('a', 7)]

In [140]: list(b.values())
Out[140]: [11, 7]

In [141]: list(b.keys())
Out[141]: ['c', 'a']
```

Conversion between list, set, tuple are trivial

```
In [142]: list(tuple(set([1, 2, 3, 2, 1])))
Out[142]: [1, 2, 3]
```

3.3.6 Object Identity

You can see whether objects are identical using the `is` operator. Or `==` on the object ID, which you can get with the `id` function.

```
In [143]: x = [1,2,3]

In [144]: y = x

In [145]: x is y
Out[145]: True

In [146]: id(x) == id(y)
Out[146]: True

In [147]: id(x)
Out[147]: 140517954138824

In [148]: y = x + [4]

In [149]: x is y
Out[149]: False

In [150]: y
```

```
Out[150]: [1, 2, 3, 4]

In [151]: y = x

In [152]: y += [4]

In [153]: x
Out[153]: [1, 2, 3, 4]

In [154]: x is y
Out[154]: True

In [155]: x = (1,2,3)

In [156]: id(x)
Out[156]: 140517998183048

In [157]: x += (4,)

In [158]: id(x)
Out[158]: 140517929751720

In [159]: x
Out[159]: (1, 2, 3, 4)

In [160]: i = 1.

In [161]: id(i)
Out[161]: 140517953992192

In [162]: i += 1

In [163]: id(i)
Out[163]: 140517954113800
```

So, we see that the inplace operator behave differently for mutable objects than for immutable objects. This is special behaviour of the *mutable* objects! Recall that strings and numbers are also immutable objects.

4 Modules and Code Organization

Besides interactive use, code can be organized in modules - python source files with extension `.py`. Many of the functions of the standard library also "live" in modules. Modules are imported using the "import" statement. As an example, let's look at mathematical functions:

4.1 Mathematical Functions

These sit in the module `math`. Functions generally are called with round brackets. To use them, we first need to import this module

```
In [164]: import math

In [165]: math.sin(12)
Out[165]: -0.5365729180004349

In [166]: math.factorial(12)
Out[166]: 479001600
```

Modules are objects. In the above example, the "." is used to access the model's `cos` function.

We can also import names for direct use

```
In [167]: from math import cos

In [168]: cos(12)
Out[168]: 0.8438539587324921
```

or even be lazy

```
In [169]: from math import atan2 as a

In [170]: a(2,3)
Out[170]: 0.5880026035475675
```

What is all in the `math` module?

```
In [171]: ? math
Type:        module
String form: <module 'math' from '/home/alex/Python/lib/python3.4/lib-dynload/math.cp
File:        /home/alex/Python/lib/python3.4/lib-dynload/math.cpython-34m.so
Docstring:
This module is always available.  It provides access to the
mathematical functions defined by the C standard.

In [172]: dir(math)
Out[172]:
(...)

In [173]: math.__dict__
Out[173]:
(...)
```

4.2 Making your Own Module

Use your favourite editor to edit the file `test.py`.

As a measure of style, we start the module - any python object - with a "doc string": On first line with a brief description, then a blank line, then the extended description on many lines and paragraphs as needed. Our code might be

```
"""
Module for python tests.

This Model cotains a selection of my python and learing codes.  It
will change a lot over time.
"""
import math

a = math.cos(12)
```

and we can use this in IPython:

```
In [174]: import test

In [175]: test.a
Out[175]: 0.8438539587324921

In [176]: test.__doc__
Out[176]: '\nModule for python tests.\n\nThis Model cotains a selection of my python

In [177]: ? test
Type:        module
String form: <module 'test' from '/home/alex/xxx/test.py'>
File:        /home/alex/xxx/test.py
Docstring:
Module for python tests.

This Model contains a selection of my python and learing codes.  It
will change a lot over time.
```

You can also add comments. Everything after a `#` symbol is treated as comment (unless in string). They can be at the end of a line, or taking up the entire line. Usually should go with the indentation of their scope

```
# I am a comment
```

4.3 Function is Form

In python code blocks are started by a colon at the end of a statement, then indentation (four white spaces) is used. The code block ends when de-indented. Lines can be continued

with a backslash at the end of the line, or when a bracket (round, square, curl) encloses an expression.

Here, let's define out first function, using the `def` statement.

```
def f(x):
    y = x + 3
    return 2*y
```

```
In [178]: import test

In [179]: test.f(3)
Out[179]: 12
```

We can also return more than one value. If there is no return statement, the default return value is `None`.

```
def f(x):
    """My test function"""
    y = x + 3
    return 2 * y, 3 * y
```

```
In [180]: import test

In [181]: test.f(3)
Out[181]: (12, 18)

In [182]: a, b = test.f(4)

In [183]: a
Out[183]: 14

In [184]: b
Out[184]: 21
```

and we can also access the function doc string

```
In [185]: ? test.f
Type:        function
String form: <function f at 0x7f3e916ba598>
File:        /home/alex/xxx/test.py
Definition:  test.f(x)
Docstring:   My test function

In [186]: test.f.__doc__
Out[186]: 'My test function'
```

Assignment to formal parameters will not overwrite actual parameter, just define inside the function to what object the name now points there. Variable defined inside the function are local and not visible outside.

```
def f(x):
    x = x + 3
    y = x + 4
    return y

In [187]: x = 5

In [188]: test.f(x)
Out[188]: 12

In [189]: x
Out[189]: 5

In [190]: test.f.y
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-28-1efa2fe5625c> in <module>()
----> 1 test.y

AttributeError: 'function' object has no attribute 'y'
```

... but we can tell python to make a variable global (to the module) and variables not local (allow assignment to enclosing scope)

```
def f(x):
    global y
    z = 0
    def g(y):
        nonlocal z
        z = 4 * x + y
        return z + 1
    y = g(2 * x) + z
    return y + 1

In [191]: import test

In [192]: test.f(2)
Out[192]: 26

In [193]: test.y
Out[193]: 25
```

In places where python expects code but you don't want to do anything, you can use the `pass` statement. Can also be used for prototyping.

```
def f(x):
    pass
```

Functions do not have to return anything, they can just perform a task, .e.g,

```
def f(x):
    print('The value is:',  x)
```

That is, they can act just like a subroutine in FORTRAN. As stated before, if there is not return value, the return `None` by default. If the return value is not used for anything, it is just ignored in codes (on the console it would be printed), even if the function does return a value.

4.4 Functions are Objects Too

```
In [194]: f = test.f

In [195]: f(2)
Out[195]: 26

In [196]: import math

In [197]: c = math.cos

In [198]: c(4 * math.pi)
Out[198]: 1.0

In [199]: x = [test.f, test.f(2)]

In [200]: x
Out[200]: [<function test.f>, 26]

In [201]: x[0](3)
Out[201]: 38
```

..and we can pass them as arguments

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

In [202]: import test

In [203]: test.f(math.cos, 0)
Out[203]: 2.0
```

or return them

```
def f(n):
    def g(x):
        return x**n
    return g

In [204]: import test

In [205]: h = test.f(3)

In [206]: h
Out[206]: <function test.f.<locals>.g>

In [207]: h(2)
Out[207]: 8
```

Note the (<locals>) above indicating a "closure" included - the function encapsulate the environment in which it was defined.

Instead of having to reload test by hand all the time, we can instruct IPython to do this automatically for us whenever the code has changed:

```
In [208]: %aimport test
```

4.5 Anonymous "Lambda" Functions

You can also define *anonymous* functions

```
In [209]: lambda x,y: x*y
Out[209]: <function __main__.<lambda>>

In [210]: f = _

In [211]: f(2,3)
Out[211]: 6

In [212]: Out[3]
Out[212]: 6
```

The second input also shows you how to capture previous output. You can also use the `%hist` magic IPython function to list previous input.

These anonymous functions are particularly useful if a function needs to be passed as argument:

```
def f(g):
    def h(x):
        return g(g(x))
    return h
```

```
In [213]: h = test.f(lambda x: x*2)

In [214]: h(3)
Out[214]: 12

In [215]: h('abc')
Out[215]: 'abcabcabcabc'
```

Note that it really just executes the operators literally, as you can see from the string example.

4.6 Function Arguments

... can have default values, and can be passed by keyword, or just by position. In either case, all positional arguments need to precede keyword arguments. Keyword arguments can be in arbitrary order, but the must not be a conflict with positional arguments when calling the function.

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

```
In [216]: import test

In [217]: test.f(2, 3)
Out[217]: 8

In [218]: test.f(y=2, x=3)
Out[218]: 9

In [219]: test.f(2, y=3)
Out[219]: 8
```

You can collect extra/remaining positional and keyword arguments in a tuple or dictionary using "*args" and/or "**kwargs" after the last positional or keyword argument, respectively. The same syntax can be used to pass such parameters from lists/tuples or dictionaries to functions. First Python will match all positional arguments, then match the keywords.

```
def f(x, *args, y = 2, **kwargs):
    return x, args, y, kwargs
```

```
In [220]: test.f(1)
Out[220]: (1, (), 2, {})

In [221]: test.f(1,2,3)
Out[221]: (1, (2, 3), 2, {})
```

```
In [222]: test.f(1, z = 4)
Out[222]: (1, (), 2, {'z': 4})

In [223]: args = dict(a = 3, b = 4, y = 5)

In [224]: kwargs = dict(a = 3, b = 4, y = 5)

In [225]: test.f(1, 2, *args, z = 3, **kwargs)
Out[225]: (1, (2, 3, 4, 5), 5, {'z': 3, 'b': 4, 'a': 3})
```

Note that keyword parameter defaults are evaluated at the time of function definition only.

5 Flow Control Structures

So far we can only do rather boring things. We want to control program flow.

5.1 If Statement

The full statement includes a required leading `if`, followed by arbitrary many `elif` and an optional final `else` statement catching the remaining cases.

```
def f(x):
    if x < 0:
        return -2 * x
    elif x == 0:
        return 0
    elif (x < 1) or (x > 5):
        x = x**0.5
        return x
    else:
        return 1
```

There is no "case" statement as in many other languages. But there is an "inline" version of the `if` statement:

```
In [226]: print('a') if 3 > 4 else print('b')
b
```

5.2 While Statement

This will continue a loop as long as a given condition is fulfilled. An `else` clause is executed if the condition test False, even if this is just the first time. `break` allows to terminate the loop, not executing the `else` clause, and a `continue` statement skips the rest and immediately continues with the next iteration test.

```
def f(x):
    y = 0
    while x > 5:
        x -= 1
        y += x
        if x > 10:
            break
        if x == 7:
            continue
    else:
        return y
    return -1
```

5.3 For Statement

This allows you to have a set number of iterations or iterate over a member of a container, etc. Similar use of `else`, `continue`, and `break`. Iterated item is assigned to a variable.

```
def f(x, n = 7):
    y = 0
    for i in range(1, n):
        y += x**i
    return y
```

and it can come from a list, dictionary, ...

```
def f(x):
    y = 0
    for i in [2, 4, 7]:
        y += x**i
    return y
```

or using enumerate to return 2-tuples of the item and its index

```
def f(x):
    d = dict()
    for i,z in enumerate(x):
        d[z] = i
    return d
```

```
In [227]: test.f('abc')
Out[227]: {'a': 0, 'c': 2, 'b': 1}
```

Another key example is the `zip` function

```
def f(x, y):
    d = dict()
    for a,b in zip(x, y):
        d[a] = b
    return d
```

```
In [228]: test.f('abc', '1234')
Out[228]: {'a': '1', 'c': '3', 'b': '2'}
```

A much larger selection can be found in the `itertools` module.

5.4 Mutants - Modifying Function Arguments

At this point I would like to note that that while you can't change what the formal parameter (from the function call) points to, the object itself that it points to, if mutable, can be changed:

```
def f(x):
    x = x + [1]
    return x
```

```
In [229]: x = [1,2,3]

In [230]: test.f(x)
Out[230]: [1, 2, 3, 1]

In [231]: x
Out[231]: [1, 2, 3]
```

but

```
def f(x):
    x += [1]
```

```
In [232]: x = [1, 2, 3]

In [233]: test.f(x)

In [234]: x
Out[234]: [1, 2, 3, 1]
```

Unless absolutely intended, this can be hard to debug and should be avoided if you can. A very popular programming style (and languages designed around it) is called **functional programming**: Functions only return values but do not modify their arguments. It is already somewhat hard to violate this in python because you can't pass parameters by

reference (as in C or FORTRAN), but in some cases you sort of can mimic this if you really want. Be careful about this when modifying mutable containers that were passed as arguments to functions!

As said, except in cases where you really know what you do and it is the most efficient way to you task and well documented: Avoid it. Especially for the API and user interfaces to your code. It is much better to use function return values or only modify objects if the API explicitly calls for and indicates this.

This can be particularly troublesome for keyword parameters with defaults, as mentioned above:

```
def f(x = []):
    x +=  [1]
    return x


In [235]: test.f()
Out[235]: [1]

In [236]: test.f()
Out[236]: [1, 1]

In [237]: test.f()
Out[237]: [1, 1, 1]
```

Ooops...

6 Iterators and Comprehensions

A key design of Python 3 is *lazy* data evaluation. Items will be produced as needed, and the same mechanism can be used to iterate over items. The `range` object is such an example. You can define you own iterators or iterator functions using the `yield` statement.

Very useful are comprehensions, sort of the and inline version of the `for` loop.

```
In [238]: [i*2 for i in range(10) if i % 2 == 0]
Out[238]: [0, 4, 8, 12, 16]
```

and can be nested ...

```
In [239]: [i*j for i in range(5) for j in range(5)
if i % 2 == 0 if j % 3 == 0]
Out[239]: [0, 0, 0, 6, 0, 12]
```

and for a rainy day, there is generator objects from comprehensions as well ...

```
In [240]: (2*i for i in range(5))
Out[240]: <generator object <genexpr> at 0x7fcce1493dc8>

In [241]: x = (2*i for i in range(5))

In [242]: list(x)
Out[242]: [0, 2, 4, 6, 8]

In [243]: list(x)
Out[243]: []
```

Iterators can be exhausted. [3]

[3] In practice, the `next` function return the next item form an iterator and raises and an `StopIteration` exceptions when there is no more items.

Part II

Advanced Python

7 Classes - The Heart of Python

In python every object also has a *type*. The type of an object is its class. You can use the `type` function to find out about the type of an object.

```
In [244]: type((1,))
Out[244]: tuple
```

You can define you own class by deriving from an exiting class - which can be one of your own classes -, hence *inheriting* its methods and attributes, but add more features and specialization. If you start a new class hierarchy from scratch, you usually would start by inheriting from `object`. But you can also inherit from multiple classes, merging their features.

```
class MyClass(object):
    """ My test class"""
    pass
```

You can now create an object of this type, i.e., an *instance* of this new class by "calling" it.

```
In [245]: o = test.MyClass()

In [246]: o
Out[246]: <test.MyClass at 0x7fcce1424630>
```

This is not yet very exciting. The most important first step is to initialize the object. This is done by the `__init__` method, to which the arguments of the object creation are passes. Additionally, a first argument `self` is passed, which is a reference to the current instance of this class and can be used to access the class's attributes and methods. Note that you can read the classes attributes but by default do not overwrite them using `self`. The `__init__` method must not return a value.

```
class MyClass(object):
    """ My test class"""
    def __init__(self, x):
        """initialize my object"""
        self.x = x
        self.y = x + 1
```

```
In [247]: o = test.MyClass(2)

In [248]: o.x
Out[248]: 2

In [249]: o.y
Out[249]: 3
```

You could also set things by hand on your class ...

```
In [250]: o.z = 4

In [251]: o.__dict__
Out[251]: {'y': 3, 'x': 2, 'z': 4}
```

... but this would be much more painful if you have to do it by hand everywhere this object is used, especially for more involved setup cases.

You can also define constants and do computations in the class body, and of course define any number of your own functions. You can even define function names by assignment, e.g., "`__iadd__ = __add__`". You can define how you objects reacts to operands, how it is printed, what its length is, how it reacts to indexing (`[]`) or being called (`()` - `__call__`).

As an example, let's have a class that stores temperature:

```
class Temperature(object):
    unit = 'K'
    def __init__(self, T = 0):
        self.set(T)
    def set(self, T):
        if isinstance(T, Temperature):
           T = T._TK
        assert T >= 0
        self._TK = T # temperature in Kelvin
    def get(self):
        return self._TK
    def e(self):
        return 7.5657e-15 * self._TK**4
    def absolute(self):
        return self._TK
    def __str__(self):
        return '{} {}'.format(self.get(), self.unit)
    __repr__ = __str__

class Celsius(Temperature):
    unit = 'C'
    offset = 273.15
    def set(self, T):
        super().set(T + self.offset)
```

```
    def get(self):
        return super().get() - self.offset
```

```
In [252]: t = test.Temperature(3)

In [253]: t
Out[253]: 3 K

In [254]: t.e()
Out[254]: 6.128217000000001e-13

In [255]: t = test.Celsius(3)

In [256]: t
Out[256]: 3.0 C

In [257]: t.absolute()
Out[257]: 276.15
```

Exercise:

1. **Add a class that deals with temperature in Kelvin.**
2. **Add function that computes the energy flux for black body radiation.**
3. **Add a functionality so two temperature objects can be added. (Consider: what happens if you add two different object types? What should the resulting object type be?)**

An example of a class of which instances behave like a function:

```
class F(object):
    def __init__(self, n):
        self.n = n
    def __call__(self, x):
        return x**self.n
```

```
In [258]: f = test.F(3)

In [259]: f(5)
Out[259]: 125
```

You can also have "computed" properties of you object defining the "`__getattr__`" method - but this is **very** advanced.

```
class X(object):
    def __init__(self, x, v):
        assert isinstance(x, str)
        self._x = x
        self._v = v
    def __getattr__(self, x):
        if x == self._x:
            return self._v
        raise AttributeError()

In [260]: x = test.X('s', 3)

In [261]: x.s
Out[261]: 3
```

8 Naming and Formatting Conventions

- use doc strings
- CamelCase for classes
- runinnames for functions and methods
- _names starting with underscore for object private methods and attributes
- use white space before and after operators and after comma.
- use 4 spaces for indentation; not less (or more) and no tabulators.

9 Making Executable Scripts

There is four things that you need.

1. A proper operating system (not Windows) - well, can be done there as well I suppose.
2. Make the python file (script) executable. On Linux we would do this on the shell

   ```
   chmod u+x text.py
   ```

   Usually this is the last step
3. Add a "shebang" (#!) at the beginning of the script. Typically I use

   ```
   #! /usr/bin/env python3
   ```

   This tells it what interpreter to use to execute your script. What you need to have there may vary from system to system. Alternatively, you can call you script later using

```
python3 test.py
```

The file does not require to retain the extension `.py`, or you can make a symbolic link (this is what I tend to do).

4. Execute specific script code if the module is called as main program. In this case the variable `__name__` contains the string `__main__`, so you start your scrip code block using the line

```
if __name__ == "__main__":
```

Other useful things are to access parameters passed to the script. These can be found in `sys.argv` - you need to import `sys` of course to use this. Note that `sys.argv[0]` is the program name. Note that the parameter are strings.

```
#! /usr/bin/env python3

"""my test script"""

import sys

a = 4

def f(x):
    print(x * a)

if __name__ == "__main__":
    f(sys.argv[1])

~/>./test.py 34
34343434
```

A very useful package for dealing with input parameters for scripts is `argparse`. Having this available is what made me switch from Python 2 to Python 3.

10 When Things Fail - Exceptions

Exceptions are a regular means of "out of band" communication in python. Some things are easiest done this way and some not really in any other way. *Use them.*

The code block to be monitored is started with a `try` statement; exceptions are "caught" by one or several `except` clauses - generally or specialized for specific exceptions, the `else` clause is executed if nothing fails and the `finally` clause is execute in any case - failure or not.

```
def f(x):
    y = 0
```

```
    try:
        for x in range(x, 3):
            y = 1 / x
    except ZeroDivisionError as e:
        print('Error:', e)
    except Exception as e:
        print('Unexpected error:', e)
    else:
        print('all went fine')
    finally:
        print(y)


In [262]: import test

In [263]: test.f(2)
all went fine
0.5

In [264]: test.f(-2)
Error: division by zero
-1.0

In [265]: test.f(-2.5)
unexpected error: 'float' object cannot be interpreted as an integer
0
```

You can also use just "`except:`" to catch all exceptions if you don't care about the exception object itself or leave off the "`as ...`" part if don't need the exception info.

```
def f(x):
    y = 0
    try:
        for x in range(x, 3):
            y = 1 / x
    except ZeroDivisionError:
        print('Error!')
    except:
        print('Unexpected error!')
    else:
        print('all went fine')
    finally:
        print(y)
```

11 Input and Output

11.1 Format Strings

Strings have a `format` method. It allows you to replace "placeholders" - in curly braces - by arguments. They can be matched by index (zero-based) or by keyword. A detailed description is found at `https://docs.python.org/3.4/library/string.html`. The layout of these placeholder is to first give what is to be formatted - either by number or by name, followed by a colon, then the format string. If no number or name is supplied, numbering is automatic. Examples:

```
In [266]: 'The Winner is {:} on {:}'.format('Mr. X', 'best movie')
Out[266]: 'The Winner is Mr. X on best movie'

In [267]: 'The Winner is {act:>10s} for A${prize:05d}'.format(prize=1000, act='Jim')
Out[267]: 'The Winner is        Jim for A$01000'
```

Another very useful string method is `join` to combine a list of strings.

```
In [268]: '/'.join(('home','alex','xxx'))
Out[268]: 'home/alex/xxx'
```

11.2 File I/O

Very useful to consequently use the routines in `os` and `os.path` to manage paths. More recently the module `pathlib` was added, but I do not have experience with it and find it sort of clunky. You may also need some routines from the `sys` module.

The `open` routine opens a file and returns a file object. To close the file, call it's `close` method. Always close you files when done. Resources are finite. `https://docs.python.org/3/library/functions.html#open`

The routine takes a file name and an open "mode". Useful modes are `t` for "text", `b` for binary, `U` for "universal new line" mode, `r` for read, `w` for write, `a` for append; `x` for exclusive creation; and added `+` opens it for "updating" (may truncate).

Use the `write` routine to write to a file, `read` to read (the entire) file, or read a single line using `readline`.

```
def f():
    f = open('xxx.txt', 'wt')
    f.write('123\n345\n5677')
    f.close()
```

Text files can be iterated over - each iteration yields one line:

```
def f():
```

```
    f = open('xxx.txt', 'rt')
    for i, l in enumerate(f):
        print('{:05d} {}'.format(i, l.strip()))
    f.close()
```

Here the `strip` method of the string gets rid of (lead/trail) which spaces and the trailing newline (`\n`).

We may deal with binary files later.

11.3 Resource Management - With Statement

The `with` statement allows you to manage resources. For example, automatically close them, and deal with exceptions (even close them then).

```
def f():
    with open('xxx.txt', 'rt') as f:
        for i, l in enumerate(f):
            print('{:05d} {}'.format(i, l.strip()))
```

12 Decorators

Start with the `@` symbol and are essentially functions that return modified "decorated" functions. You may see this sometimes.

```
def mul5(f):
    def g(*args, **kwargs):
        print('function was decorated.')
        return f(*args, **kwargs) * 5
    return g

@mul5
def h(x):
    return x**2
```

```
In [269]: test.h(3)
function was decorated.
Out[269]: 45
```

The key point is that the decorator takes a function and returns another function.

The decorator itself can also be a function - can have parameters - that returns the actual wrapping function.

```
def multiply(x):
    def wrap(f):
```

```
        def g(*args, **kwargs):
            print('function was decorated:', x)
            return f(*args, **kwargs) * x
        return g
    return wrap

@multiply(5)
def h(x):
    return x**2


In [270]: test.h(3)
function was decorated: 5
Out[270]: 45
```

Or you can design it as a class with a `__call__` method that takes parameters for its `__init__` method.

```
class Multiply(object):
    def __init__(self, factor):
        self._factor = factor
    def __call__(self, f):
        def g(*args, **kwargs):
            print('function was decorated:', self._factor)
            return f(*args, **kwargs) * self._factor
        return g

@Multiply(5)
def h(x):
    return x**2
```

13 Properties

Allows you to provide an interface to internal data of you object. You can generally control object access using `__get__` and `__set__`.

```
class Temperature(object):
    def __init__(self, T = 0):
        self.T = T
    @property
    def T(self):
        """T in K"""
        return self._TK
    @T.setter
    def T(self, T):
        self._TK = T
    def e(self):
```

```
        """energy density in cgs"""
        return 7.5657e-15 * self._TK**4

class Celsius(Temperature):
    offset = 273.15
    @property
    def T(self):
        """T in C"""
        return self._TK - self.offset
    @T.setter
    def T(self, T):
        self._TK = T + self.offset


In [271]: t = test.Celsius()

In [272]: t.T
Out[272]: 0.0

In [273]: t = test.Celsius()

In [274]: t.T = 100

In [275]: t._TK
Out[275]: 373.15

In [276]: t.T
Out[276]: 100.0
```

using the `property` function you can also define properties that can only be set but not read.

```
class Temperature(object):
    def __init__(self, T = 0):
        self.T = T
    @property
    def T(self):
        """T in K"""
        return self._TK
    @T.setter
    def T(self, T):
        self._TK = T
    def e(self):
        """energy density in cgs"""
        return 7.5657e-15 * self._TK**4

class Celsius(Temperature):
    offset = 273.15
    @property
```

```
    def T(self):
        """T in C"""
        return self._TK - self.offset
    @T.setter
    def T(self, T):
        self._TK = T + self.offset
    def _set_TF(self, T):
        self.T = (T - 40) * 5 / 9
    TF = property(fset = _set_TF)

In [277]: t = test.Celsius()

In [278]: t.TF = -20

In [279]: t.T
Out[279]: -33.33333333333334

In [280]: t.TF
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-21-ed8d890034be> in <module>()
----> 1 t.TF

AttributeError: unreadable attribute
```

14 Regular Expressions

Knowing how to use regular expressions will make your life much easier. Yes, it requires some work to get started. Python offers some powerful tools to use them for you text processing, e.g., extracting data from text files or web pages. The manual is at `https://docs.python.org/3.4/library/re.html` The main use it to match items that have certain patterns.

Key tokens are `(` and `)` to "capture" strings you want to extract, `.` to match anything, `?` to match the previous item zero to one times, `*` to match the previous item any number of times, `+` to match the previous item at least once, `^` for beginning of string/line, `$` for end of string/line, `[` and `]` to define a set of characters, etc. You also have special tokens like `\d` matching a digit, etc.

```
In [281]: re.findall('([123]+)', '1234ghgs7sjj4399')
Out[281]: ['123', '3']
```

An example script I have used to renumber all the input/output prompts from IPython to be consecutive in the script.

```
#! /usr/bin/env python3
```

```
import sys, re, os

def format(infile):
    outfile = infile + '.tmp'
    In  = re.compile('^(In \[)[0-9]+(\]:.*)')
    Out = re.compile('^(Out\[)[0-9]+(\]:.*)')
    Search = (In, Out)
    count = 0
    with open(infile, 'rt') as f, open(outfile, 'xt') as g:
        for line in f:
            for prompt in Search:
                 m = prompt.findall(line)
                 if len(m) == 0:
                     continue
                 if prompt is In:
                     count += 1
                 line = prompt.sub(r'\g<1>{:d}\g<2>'.format(count),
                                   line)
            g.write(line)
    os.remove(infile)
    os.rename(outfile, infile)

if __name__ == "__main__":
   format(sys.argv[1])
```

15 NumPy

A very widely used convention is to import `numpy` as `np`

```
import numpy as np
```

If you start IPython with the `--pylab` flag, it will do this automatically, however, in you scripts you still have to do it by hand.

NumPy provides multi-dimensional array class, `ndarray`, optimized for numerical data processing. It allows various data types - it defines its own data types, called "`dtypes`", and even has record arrays. I recommend you the very good online documentation at `http://www.numpy.org/`.

There is too much to tell about NumPy to fit into the time allocated for this course. Some of the key features are to allow you to do indexing nut just in the slice notation, but also by lists of indices or multi-dimensional constructs, or by arrays of truth values. The latter is extremely useful to avoid loop with if statements.

Key advice: Use "advanced slicing" to replace loops with if statements! These are very slow. Should this really not be possible to avoid, you can easily write FORTRAN extension modules using `f2py`.

Examples:

```
In [282]: x = np.arange(12).reshape(3, -1)

In [283]: x.shape
Out[283]: (3, 4)

In [284]: x[0,2]
Out[284]: 2

In [285]: x[2]
Out[285]: array([ 8, 9, 10, 11])

In [286]: x
Out[286]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [287]: ii = x % 2 == 1

In [288]: ii
Out[288]:
array([[False,  True, False,  True],
       [False,  True, False,  True],
       [False,  True, False,  True]], dtype=bool)

In [289]: x[ii] = 0

In [290]: x
Out[290]:
array([[ 0,  0,  2,  0],
       [ 4,  0,  6,  0],
       [ 8,  0, 10,  0]])

In [291]: y = np.array([1, 2, 3, 4])

In [292]: y
Out[292]: array([1, 2, 3, 4])

In [293]: x *= y

In [294]: x
Out[294]:
array([[ 0,  0,  6,  0],
       [ 4,  0, 18,  0],
       [ 8,  0, 30,  0]])

In [295]: y = np.array([1,2,3])

In [296]: x *= y[:, np.newaxis]
```

```
In [297]: x
Out[297]:
array([[ 0,  0,  6,  0],
       [ 8,  0, 36,  0],
       [24,  0, 90,  0]])
```

The example above shows NumPy by default tries to match the last dimension and automatically expands the others. To change that, you can add "extra axes" using `np.newaxis`.

Telling you all about `NumPy` would likely be an entire course by itself.

16 Matplotlib

Matplotlib is a frequently used Python package for plotting. To use it interactively on the IPython shell, use the `--pylab` flag when starting IPython. Much of the interactive plotting interface is quite similar to Mathlab, so it may not seem all that strange. Matplotlib is fully object-oriented, and so is the graphics it produces: Objects can be modified, even interacted with, e.g., react to users clicking at them (advanced programming, I have never used).

Whereas you can do things interactively on the shell for development - I do that, in part - I highly recommend that you put the finalized scripts inside a script, possibly a function, but surely preferable an object. Even it you just use the `__init__` function for all the plotting at first - later you can delegate some of the tasks, e.g., setups you frequently use, into separate routines, maybe in a base class ("`MyPlot`") from which you derive specialized plots. You can store properties of the plot, the figure object, axes objects, etc. in the class and later access them

The key is that having plots in scripts, you can easily modify things and rerun the script, or if your data or model has changed, you can just re-run the script - especially for last minute changes when you thesis is due (advisor says: "Change the color of that line!").

There is a vast variety of different plot and recipes to make them. A prominent gallery with code examples can be found at `http://matplotlib.org/gallery.html`. Personally, I don't think this is always done very well, but it is a good start. I highly recommend you have a look at the Beginner's and Advanced guides (`http://matplotlib.org/devdocs/contents.html`).

As an example, let's make a simple line plot:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import os.path

projectpath = os.path.expanduser('~/xxx')
```

```
class MyPlot(object):
    def __init__(self, func = lambda x: np.sin(x)**2):
        f = plt.figure()
        ax = f.add_subplot(1, 1, 1)
        x = np.linspace(1, 10, 100)
        y = func(x)
        ax.plot(x, y,
                color = 'r',
                lw = 3,
                label = 'Model A')
        ndata = 10
        x = np.random.choice(x, ndata)
        y = func(x) + np.random.rand(ndata) * 0.1
        ax.plot(x, y,
                color = 'b',
                marker = '+',
                linestyle = 'None',
                markersize = 12,
                markeredgewidth = 2,
                label = 'Data')
        ax.set_xscale('log')
        ax.set_xlabel(r'$x\,(\mathrm{cm})$')
        ax.set_ylabel(r'$\sin^2\left(x\right)$')
        ax.legend(loc='best')
        f.tight_layout()
        plt.show()
        self.f = f

    def save(self, filename):
        self.f.savefig(os.path.join(projectpath, filename))
```

```
In [298]: p = test.MyPlot()
```

```
In [299]: p.save('xxx.pdf')
```

Matplotlib will automatically create an output file format based on the file extension.