A Whirlwind Tour of Programming in Python

Peter K. G. Williams (CfA) • @pkgw • http://newton.cx/~peter/

CHAMP Camp · Pomona, CA, USA · 2017 Jun 13

1. Foundations

Code is math.

Programs perform *computations* by implementing *algorithms*.

It is possible for algorithms to be *provably correct* and to describe precisely how efficient they are.

It is possible to write programs that *never fail*.

Algorithms need not execute on CPUs. E.g., knitting.

Code is engineering.

Programs are written to perform useful tasks.

They are produced in the context of requirements, schedules, and budgets.

The justifiable investment in a program scales with how much value it generates: how often it is run, the costs if it fails.

(This is why we're teaching you Python 2 and not Python 3.)

Code is a language.

There are both syntax and semantics.

There are idioms and clichés.

You gain fluency over time.

You won't learn to write well without reading a lot! A *lot*.

Code is an edifice of human culture.

Every program exists in some engineered context: a CPU, knitting needles and yarn, ...

These contexts are as rich as any other human endeavors: they have histories, factions, fads, masterpieces, and so on.

It is hard to say how important it is to understand this context.

But: bad programs often go “against the grain” of the context in which they operate.

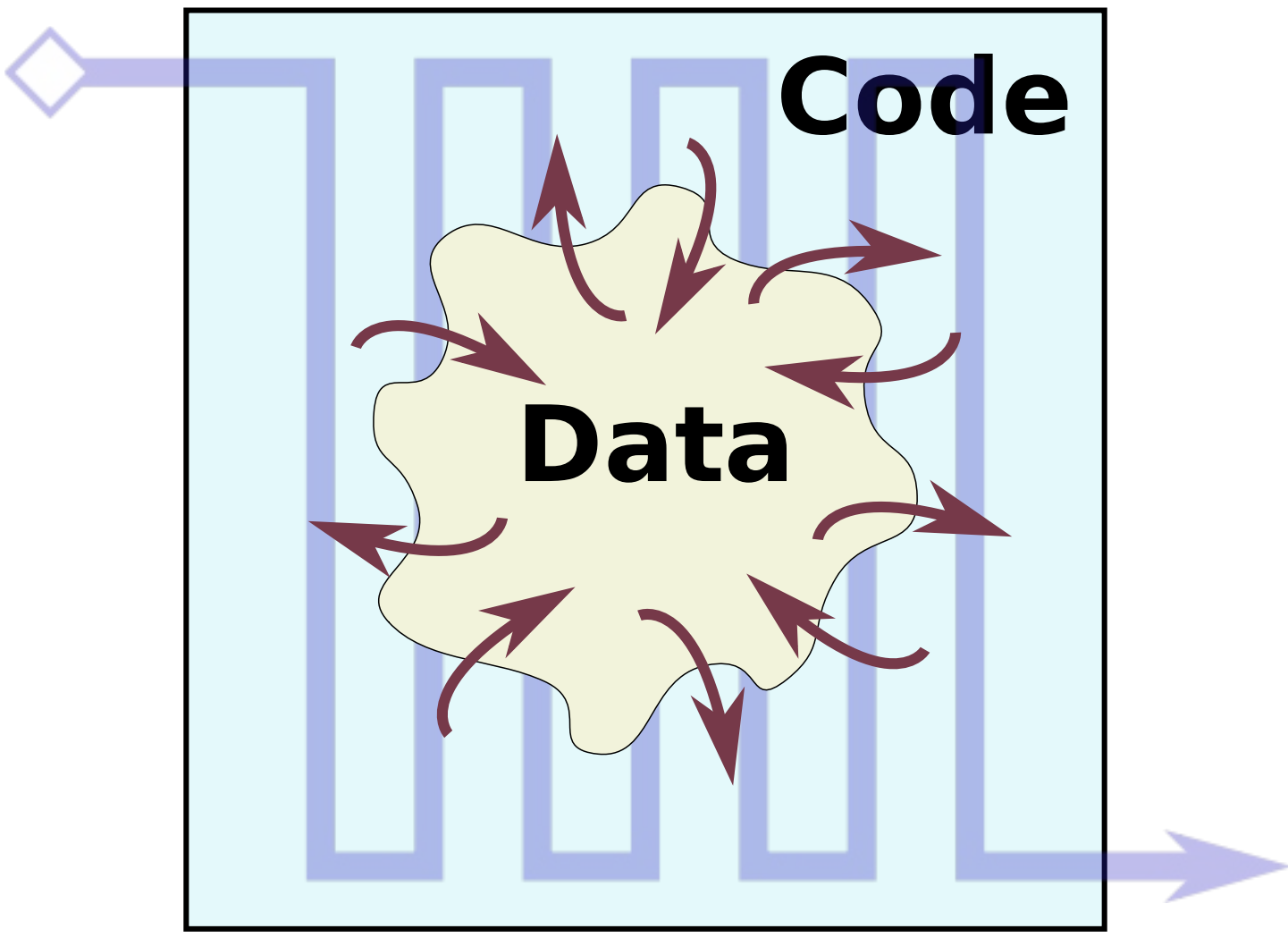
Why do we write code?

1. To *automate* and *accelerate* tedious tasks.
2. To make computations *reproducible* and *auditable*.

2. Data Structures

Programs are made of code and data.

There is as much subtlety in designing your data as there is in your code!



Every language has types of data.

Python provides “core” types:

- The “None” type: an empty or undefined value
- Booleans (“bools”): true/false values
- Integers
- “Floats”: approximations of real numbers
- “Strings”: text — sequences of characters

It also has *collections* for making *data structures*:

- Lists
- Sets
- Dictionaries (“dicts”)
- *Objects*, to be explored later

Variables are names for data.

Variable assignment in Python is totally standard, and also totally weird if you're not used to it:

```
x = 1
```

In Python it is important to distinguish between a piece of data itself and the name used to refer to it.

```
a = ['list', 'of', 'words']
b = a
b[1] = 'new'
print a[1]
```

Indirection is tricky and comes up *all over*.

Mutable data have pitfalls.

The trip-up in the previous slide only happened because the data underlying *a* could be modified.

Mutations are relatively easy to keep track of in simple, linear programs.

Avoiding mutability makes it *much* easier to build complex software systems.

3. Control Flow

Real programs don't run linearly.

Our first examples ran commands strictly in sequence.

Most real programs have to make choices or run loops. The geometry of these *branches* define its *control flow*.

Python provides a very standard set of statements for managing control flow.

if **statements are universal.**

They only run a certain block of statements if a certain expression evaluates to a “truthy” value.

```
if x > 4:
    x = 4
    print 'Capped x.'
print x
```

Python groups statements into *blocks* by looking at their indentation!!! Most languages don’t.

if **statements are universal.**

Full generality:

```
if {condition-1}:
    {block-1}
elif {condition-2}:
    {block-2}
...
elif {condition-n}:
    {block-n}
else:
    {block-(n+1)}
```

while **loops are also universal.**

Example:

```
n_iterations = 0
while n_iterations < 10:
    do_some_stuff()
    n_iterations += 1
print 'Finished iterating.'
```

while **loops are also universal.**

Example:

```
total = 0
while True:
    x = get_next_x()
    if x < 0:
        print 'Quitting'
        break
    if x > 9:
        print 'Ignoring'
        continue
    total += x
print total
```

for **loops are for containers.**

Example:

```
a_list = [1, 1, 2, 3, 5]

for x in a_list:
    print x
```

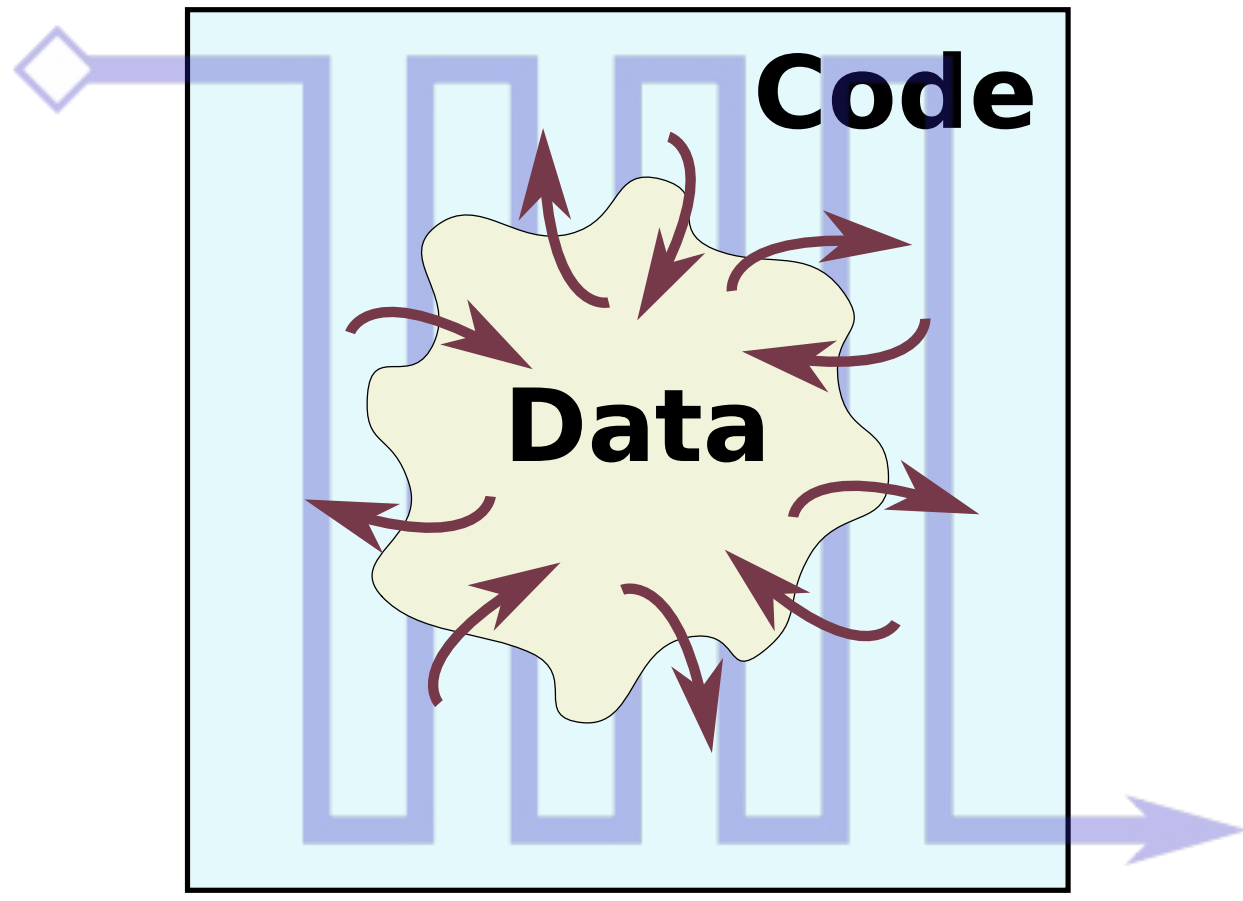
```
a_dict = {'key1': 8, 'key2': 13}

for k in a_dict.keys():
    print k
```

4. Input and Output

IO and computation are fundamentally different.

Recall the code/data schematic ⇒



Computation happens inside the box.

IO reaches outside the box. It is slow and errors are *always* possible.

But: every program needs IO.

There are many kinds of IO.

- Files on disk
- Output to the terminal
- Network communications
- Talking to hardware

The OS provides abstractions so that you can use the same techniques to work with all of these.

But often you need to understand how they work under the hood.

Low-level IO centers on byte streams.

(*Byte*: ordered group of eight booleans [bits]
⇒ 2^8 possible combinations
⇒ interpretable as an integer between 0 and 255.)

Stream, empirically: something that lets you read or write a "next" batch of bytes.

Often valuable, and sometimes easier, to process data in a streaming paradigm.

But: when reading and writing data, use higher-level abstractions!

Python has many ways to do IO.

Special data — *file-like objects* — represent byte streams.

Simplest way to do output: print statement for *line-oriented text*.

Beyond that: *there's no simple answer*. But there are many options. Use them instead of rolling your own!

5. Managing Complexity: Functions & Modules

Managing complexity is hard.

Modern programming languages don't just give you tools to perform computations and IO.

They spend just as much effort to give you tools to manage the complexity of large programs.

The name of the game is breaking a big task into smaller, *composable* pieces.

Much of "learning how to code" is learning how to divide problems into solvable sub-problems.

Functions help manage complexity.

```
def factorial(n):
    if n == 0: # ← equality test
        return 1
    return n * factorial(n - 1)

print factorial(5)
```

Functions are tools for:

1. *Encapsulation*
2. Preventing *code duplication*
3. Control flow management

Functions are variables too.

We lied: code is data.

```
def make_a_multiplier(k):
    def multiply_by_k(x):
        return k * x
    return multiply_by_k

multiply_by_5 = make_a_multiplier(5)
print multiply_by_5(3)
```

Organize Python code with modules.

Modules encapsulate related code and data.

A module is a single file of Python code.

When you *import* a module, the code is run and the resulting variables are made available to you.

You can’t import two “different versions” of the same module.

Modules are also variables.

Their contents are once again accessed with “dot notation”.

```
import sys
print sys.path
also_sys = sys
```

```
import os.path
import numpy as np
from astropy.io import ascii, fits
```

Modules are for code distribution too.

Virtually all modern software relies on *library* code.

Virtually all modern languages consist of a *core* plus a *standard library*.

Python libraries are distributed as *packages* providing one or more modules.

Python's package ecosystem is ungainly, but there's a package for everything.

The *pip* and *conda* tools are similar but different. It is important to learn how to use them.

6. Managing Complexity: Classes

You can create your own data types.

You can define a new *class* that combines data and methods.

You can then create one or more *instances* of that class.

Anything that can be assigned to a variable in Python is an instance of some class.

Here's a typical class definition.

```
class Counter(object):
    value = 0

    def increment(self):
        self.value += 1

    def get(self):
        return self.value

my_counter = Counter()
my_counter.increment()
print my_counter.get()
```

Custom types are a complexity tool.

Classes are like “mini modules”: a tool for encapsulating related code and data.

Also like a module: all of the code inside the **class** block is run, and then the resulting assignments are bundled up and made available.

Key difference: you can have multiple instances of a class.

Classes often have constructors.

This is usually problematic:

```
class Counters(object):
    values = [0, 0, 0]
    ...
```

Instead:

```
class Counters(object):
    def __init__(self, n_items):
        self.values = [0] * n_items

    ...
```

Python's class system is elaborate.

For instance, it has *inheritance*, where one class derives from another:

```
class MyDictionary(dict):
    ...
```

You will not often need to use inheritance.

There are many magic functions: __eq__, __getattr__, ...

As ever: at some point you should read the detailed rules.

Manage statefulness in your classes.

Classes make it easy to introduce *hidden state*.

```
class MyClass(object):
    mood = 'nice'
    def turn_mean(self):
        self.mood = 'mean'
    def do_something(self):
        if self.mood == 'nice':
            print 'Hello!'
        else:
            sys.delete_all_files()
...
my_instance.do_something() # ????
```

7. Error Handling

There are four kinds of errors.

1. Syntax errors
2. Implementation errors
3. Design errors
4. Environmental errors

A perfect coder could avoid the first three, but *all* real programs are subject to environmental errors.

Python uses an exception paradigm.

Errors cause the control flow to jump to special *except blocks* that you can provide.

If no appropriate except block is available, the program aborts.

This paradigm is weak because it makes error handling very implicit, but it has its virtues.

Python exceptions are data too.

They are instances of the Exception class, or one of its inherited *sub-classes*.

```
try:
    do_some_io(file_name)
    do_more_io(file_name)
except IOError as e:
    print 'Uh-oh:', (e, file_name)
    raise

print 'We made it to here.'
```

Don’t “handle” an error by ignoring it.

It is tempting to write:

```
try:
    big_chunk_of_code()
except Exception as e:
    print 'Error:', e
```

DO NOT DO THIS!!!!

Exceptions signify *major failures*. Only ignore them in *tightly-controlled conditions*.

Exceptions come with lots of meta-data. Printing them yourself loses the information.

Handle exceptions narrowly.

Isolate the specific lines that might fail, and catch a specific error type:

```
try:
    import scipy
    have_scipy = True
except ImportError:
    have_scipy = False
```

Fix your bugs.

Try/except statements are for environmental errors: the ones you can't prevent.

Solve other kinds of errors by *fixing your code*! Or by using more appropriate tools. Not:

```
try:
    v = mydict[keyname]
except KeyError:
    v = 0
```

Instead:

```
v = mydict.get(keyname, 0)
```

Turn exceptions into error codes.

Exception data do not “serialize” well.

When recording whether an operation was successful or not, recall *Anna Karenina*:

“Happy families are all alike; every unhappy family is unhappy in its own way.”

A small taxonomy of errors is usually useful:

```
SUCCESS = 0
CONVERGENCE_ERROR = 1
IO_ERROR = 2
...
```

8. Conclusion

Recap.

We've covered a lot of ground today!

- Basic data types
- Conditionals, loops
- Input and output
- Functions
- Modules
- Classes
- Exceptions

Some parting thoughts.

As you learn to code, there will be frustrating days.

Don’t feel bad about that! Software is subtle and there are lifetimes’ worth of ideas to master.

Don’t cut us any slack, either! It is harder to learn to code than it should be, and we should fix that.

Thanks for your attention!

Peter K. G. Williams
@pkgw • http://newton.cx/~peter/

HTML talk info: http://tinyurl.com/htmltalk • *Design credits*: Hakim El Hattab (“white” theme), Julieta Ulanovsky (Montserrat font), Steve Matteson (Open Sans font) • *Tech credits*: git, reveal.js, Firefox developer tools.