Argument against Python in Education

Principal Author: Patricio Whittingslow

Authors: Mario Rugiero

Table of contents

Introduction

This document lists evidence against using Python to teach undergraduate/graduate students and presents alternative programming languages as a solution to this problem.

The author teaches a course called "*Introducción al Pensamiento Computacional*" (Introduction to Computational Thinking in English). This is an undergraduate course that introduces the following concepts to students who may have never programmed before:

- Expressions
- Basic data structures (Python primitives, dict, list, tuple)
- Control structures (if, while, for)
- Functions

The course also deals with introduction to data analysis using pandas**,** numpy and matplotlib libraries.

Throughout this document the author will refer to "**modern languages**". This makes reference to a recent change in language design preference which led to various general purpose programming languages which are memory safe, statically typed and apt for system programming such as Scala, Swift, D, Rust, Go, Zig, Nim, Kotlin among various others.

Motivation

In 2020 a study was performed to evaluate what skills help students learn languages faster. This study was performed with Python as the language of choice. The results of the study were astounding to the programming world. Math was not a skill needed to be a programmer, predicting only 2% of the variance. The headline in many blog posts treating the subject reads "*Learning to code requires language skills, not math*". This is because natural language skills (reading normal text) predicted 17% percent of the variance. This result put newfound interest in the importance of code legibility [8].

What was not talked about as much were the other predictors. Working memory capacity and reasoning skills predicted a staggering 34% of the variance. This particular result is of interest to the author precisely because the study was performed with Python. Maybe this predictor can be reduced if Python is swapped out for an easier language to learn. And if this can be done, it would make the field of programming so much more accessible to those who do not have the best of reasoning skills.

Outline of Problems

Problems described in detail throughout this document are summarized briefly below.

- Readability: Much of Python in the wild is not readable, this includes the standard library. Students then miss out on reading other people's code, an important part of programming.
- Autoformatting: Boon to readability. Not present by default
- Whitespace is code: Invisible syntax has been problematic for students
- Exceptions: Students have to deal with an extra control flow concept or avoid using throwing functions like str or list.index
- Complex API: Asking students to depend on pandas/numpy offline/online documentation is unthinkable. Even standard library documentation is hard to follow for a newbie: see `open`. This forces students to constantly depend on Google search for finding examples+StackOverflow– this context switching is known to hamper learning.
- Intellisense is weak: Dynamic typing weakens dev experience. Students have less confidence due to this.
- Library cognitive overload: pandas and numpy are basically DSLs, each with their own programming paradigm
- Inconsistency: Python is an old language which means we are stuck with some design decisions.
- Python obscures data: Python mixes behavior and data structure in code making it harder to reason about a problem. Ideally data should be understood before solving the problem.
- No user types: Students program in the dark. There is little to no knowledge of the shape of the data when programming in Python. One must depend on the dict and list types which offer little introspection into the data.
- Python provides many ways of doing the same thing. Learning multiple ways to do one thing is harmful to the learning process [6]
- Loss of context in code: It is sometimes outright impossible to make out what data one is working with without exploring the data pipeline. This slows students down when programming. This also slows students down when reading code.

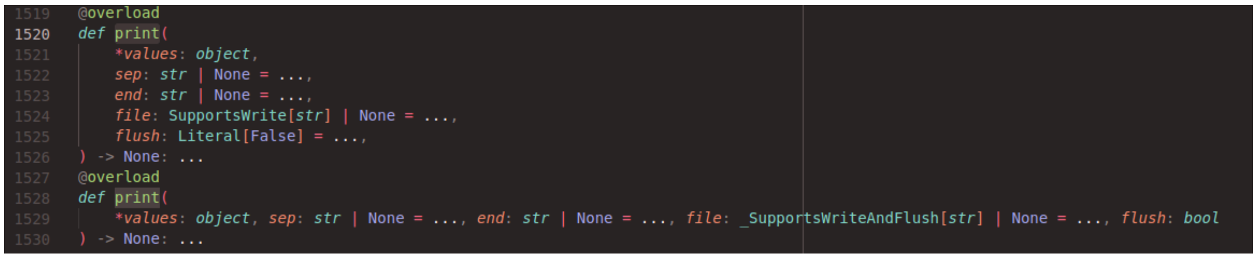
Problems

It is hard to learn

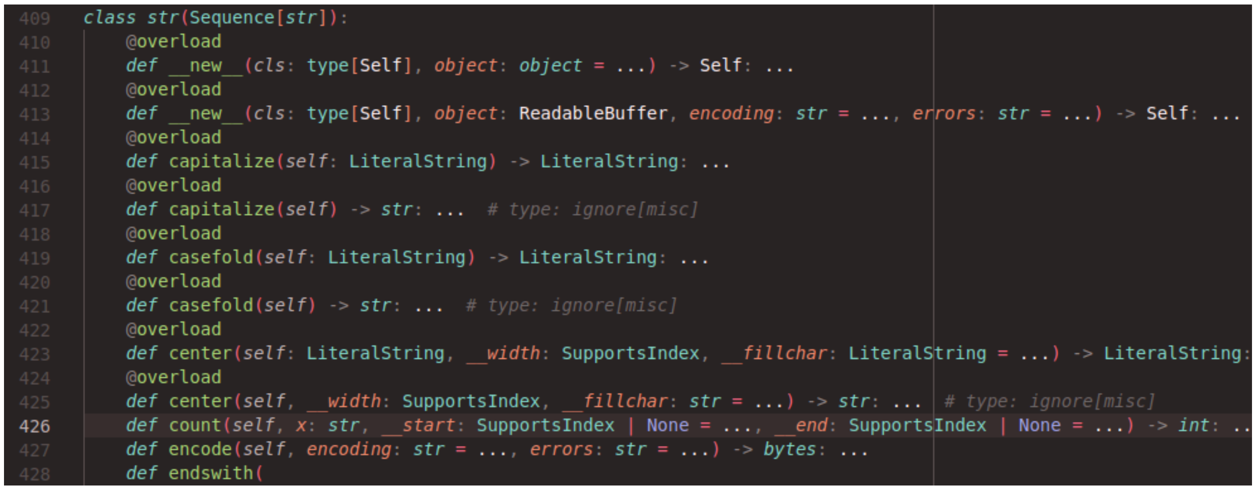
Python is often cited as an easy to learn language though the author believes this is a mixup with the idea that *Python is easy to use*. Python compared to other languages as an introductory language is notoriously hard. The main talking points on this matter are as follows.

Readability

Python as a language is widely regarded as having clean syntax and a prose-like aspect to its design. However, would you suggest a novice programmer read other people's Python code online? How about the standard library? Surely Python behind the scenes is readable?



```
@overload
def print(
    *values: object,
    sep: str | None = ...,
    end: str | None = ...,
    file: SupportsWrite[str] | None = ...,
    flush: Literal[False] = ...,
) -> None: ...
@overload
def print(
    *values: object, sep: str | None = ..., end: str | None = ..., file: _SupportsWriteAndFlush[str] | None = ..., flush: bool
) -> None: ...
```



```
class str(Sequence[str]):
    @overload
    def __new__(cls: type[Self], object: object = ...) -> Self: ...
    @overload
    def __new__(cls: type[Self], object: ReadableBuffer, encoding: str = ..., errors: str = ...) -> Self: ...
    @overload
    def capitalize(self: LiteralString) -> LiteralString: ...
    @overload
    def capitalize(self) -> str: ...  # type: ignore[misc]
    @overload
    def casefold(self: LiteralString) -> LiteralString: ...
    @overload
    def casefold(self) -> str: ...  # type: ignore[misc]
    @overload
    def center(self: LiteralString, __width: SupportsIndex, __fillchar: LiteralString = ...) -> LiteralString:
    @overload
    def center(self, __width: SupportsIndex, __fillchar: str = ...) -> str: ...  # type: ignore[misc]
    def count(self, x: str, __start: SupportsIndex | None = ..., __end: SupportsIndex | None = ...) -> int: ..
    def encode(self, encoding: str = ..., errors: str = ...) -> bytes: ...
    def endswith(
```

Truth be told, reading Python is only appealing when it's pure Python written by an experienced Pythonista.
A lot is lost when students are unable to read other people's code

- They must take our word that what they are learning is useful and solves real problems
- Reading the standard library implementation of a function (or any code in the wild) might give insight to how another person thought of the problem

Autoformatting as we will see in the following section adds readability.

Autoformatting

There has recently been a push to enforce the use of an uncompromising auto-formatter for most languages. This move began in the 1980's with Unix indent tool and is now ubiquitous in nearly every language due to the **huge** readability and productivity gains; there's even one for Python.

Even if Python remains the de facto language to be used, students should be forced to use an auto formatter that triggers on-save in the IDE. Some benefits:

- Students worry less about formatting their code, more on solving the problem. The mental energy this saves is noticeable.
- Source code now looks the same regardless of who wrote it- students can look at a friend's code and understand it immediately. Help between students should become more prevalent.

Whitespace is code

The arguments for whitespace as part of the language syntax are as follows:

1. Code ease of readability compared to curly-brace languages
2. Bugs caused by whitespace interpretation in C
3. It forces you to format your code so it is more readable

With regards to Readability

As for the first argument (1.) there is no solid evidence to support it. The author suggests after extensive experience with Python and curly brace languages with enforced auto formatters there are advantages to **readability** to each language in how it deals with control structure delimitation but that there is no language that does it all perfectly for every case.

The author also dismisses the last two arguments (2. and 3.) as invalid since most modern languages do not rely on whitespace for syntax and since most, if not all, modern languages use autoformatters which enforce not only code indentation but several other readability rules.

With regards to user experience

There is however, the issue of how whitespace as code is detrimental to the learning experience. In the author's experience it was a source of confusion to students who intuitively wrote code which **seemed** correct but were thrown off by the level of indentation. Two issues were prevalent:

- rogue whitespace in the form of a single or two spaces which crashed the python interpreter
- Indentation level mismatch with control structure

This problem persisted well into the third week after learning if statements. Other languages deal with this issue by using curly braces. There is no ambiguity when using curly braces and it follows from other teachers' experience that this is intuitive and it is not questioned nor does it bring up problems even in the case of eleven-year olds[1], as does Python's take on indentation.

Exceptions

Which operation does the reader believe is more error prone?

1. "Hello".index(character)
2. list[i:n:2]

Incredibly, Python can only throw an exception in the first case, and that would be the most likely outcome since if the character is not in the string there will be an exception.

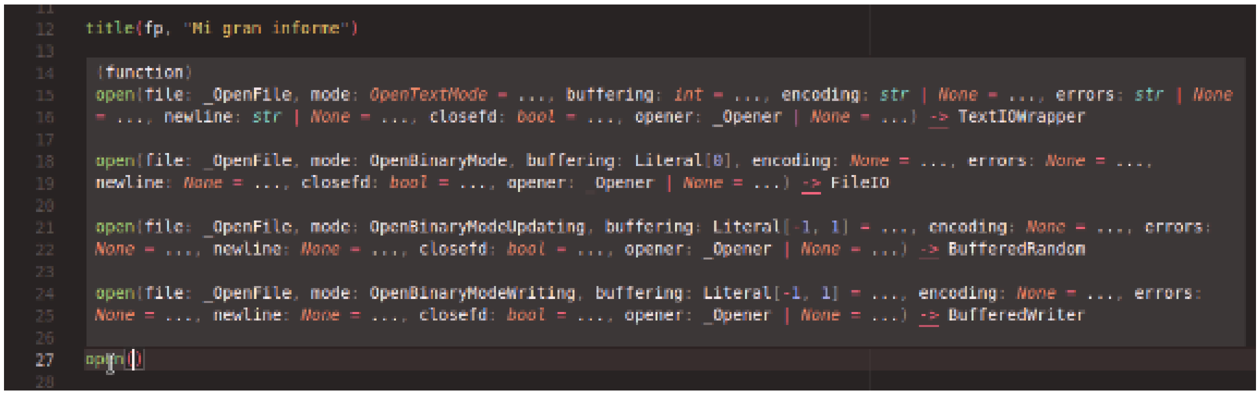
The author would like to take a moment and deviate from the point they are trying to make to show and make a point on this ridiculous aspect of Python: most if not all programmers will know the pains of off-by-one errors– it is no secret that a large portion of security vulnerabilities are due to this fact (buffer overflows). Python's response to this is "letting it happen silently" when an index is accidentally negative while raising an exception when a substring is not found in a string.

Exceptions are commonplace throughout Python and they limit the software students may write before learning "try/catch" (most introductory courses don't even teach "try") since they must think of all edge cases so that their program will not crash. This is burdensome work and completely unnecessary from the point of view of anyone who has used languages with robust error handling, monads, maybe's etc.

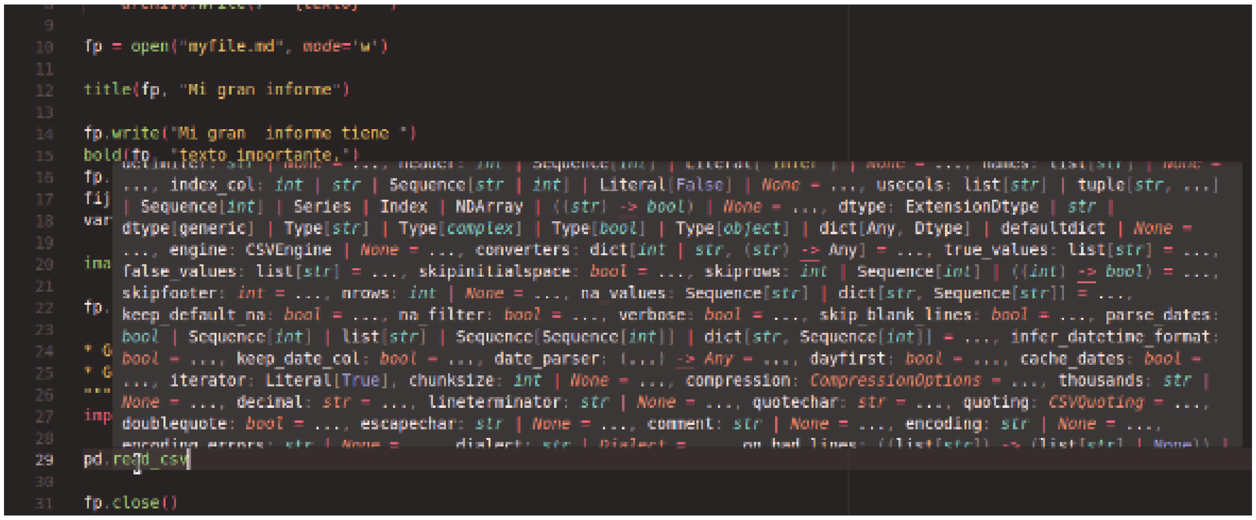
Complex APIs

The author will make reference to other APIs in this section for comparison since "Complex" is a relative term.

There are 7 ways of calling the **open** function with 7 different results in Python. The documentation is 2 pages long (11pt font) and has a table of characters to denote different ways of formulating a very important argument to open. For context on why this is "complex": compare this with Go's approach: Go provides two dedicated functions for the most common file operations, to read one uses os.Open, to create a new file for writing one uses os.Create. If one desires the generalized open functionality, one can use os.OpenFile, which has similar functionality to Python's open. In all of the cases above Go returns the same two values, a File and an error and the documentation does not exceed a paragraph.



```
title(fp, "Mi gran informe")

 (function)
 open(file: _OpenFile, mode: OpenTextMode = ..., buffering: int = ..., encoding: str | None = ..., errors: str | None
 = ..., newline: str | None = ..., closefd: bool = ..., opener: _Opener | None = ...) -> TextIOWrapper

 open(file: _OpenFile, mode: OpenBinaryMode, buffering: Literal[0], encoding: None = ..., errors: None = ...,
 newline: None = ..., closefd: bool = ..., opener: _Opener | None = ...) -> FileIO

 open(file: _OpenFile, mode: OpenBinaryModeUpdating, buffering: Literal[-1, 1] = ..., encoding: None = ..., errors:
 None = ..., newline: None = ..., closefd: bool = ..., opener: _Opener | None = ...) -> BufferedRandom

 open(file: _OpenFile, mode: OpenBinaryModeWriting, buffering: Literal[-1, 1] = ..., encoding: None = ..., errors:
 None = ..., newline: None = ..., closefd: bool = ..., opener: _Opener | None = ...) -> BufferedWriter

open()
```



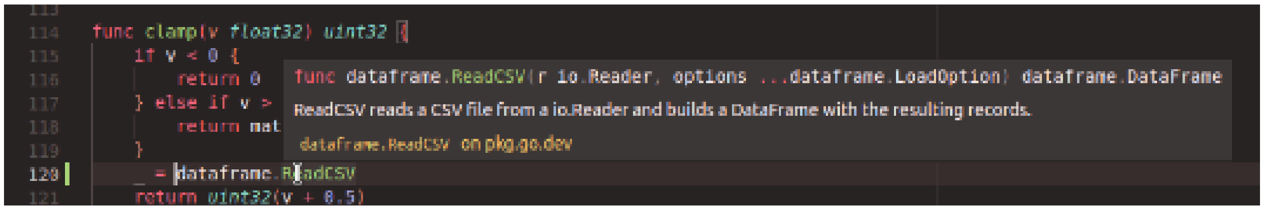
```
fp = open("myfile.md", mode='w')

title(fp, "Mi gran informe")

fp.write("Mi gran  informe tiene ")
bold(fp, "texto importante.")
fp.
fij
var

ima

fp.

* G
* G
"""
imp
pd.read_csv

fp.close()
```

```
..., index_col: int | str | Sequence[str | int] | Literal[False] | None = ..., usecols: list[str] | tuple[str, ...]
| Sequence[int] | Series | Index | NDArray | ((str) -> bool) | None = ..., dtype: ExtensionDtype | str |
dtype[generic] | Type[str] | Type[complex] | Type[bool] | Type[object] | dict[Any, Dtype] | defaultdict | None =
..., engine: CSVEngine | None = ..., converters: dict[int | str, (str) -> Any] = ..., true_values: list[str] = ...,
false_values: list[str] = ..., skipinitialspace: bool = ..., skiprows: int | Sequence[int] | ((int) -> bool) = ...,
skipfooter: int = ..., nrows: int | None = ..., na_values: Sequence[str] | dict[str, Sequence[str]] = ...,
keep_default_na: bool = ..., na_filter: bool = ..., verbose: bool = ..., skip_blank_lines: bool = ..., parse_dates:
bool | Sequence[int] | list[str] | Sequence[Sequence[int]] | dict[str, Sequence[int]] = ..., infer_datetime_format:
bool = ..., keep_date_col: bool = ..., date_parser: (...) -> Any = ..., dayfirst: bool = ..., cache_dates: bool =
..., iterator: Literal[True], chunksize: int | None = ..., compression: CompressionOptions = ..., thousands: str |
None = ..., decimal: str = ..., lineterminator: str | None = ..., quotechar: str = ..., quoting: CSVQuoting = ...,
doublequote: bool = ..., escapechar: str | None = ..., comment: str | None = ..., encoding: str | None = ...,
```

Above is an image of the arguments to **pandas.read_csv**. When a student sees the argument list, they probably "nope" out, as in ignore it as important. Surely something as complex as what's seen above is not something the student should know how to use. Due to this unfortunate API design students stop depending on intellisense/documentation and prefer using stack overflow and other online resources. This is a **slow** way to learn.

- Students lose context when switching from IDE to browser. Depending on how well the students can refocus and regain context, time may be lost when switching back to the IDE
- Intellisense has more context on the API than the student. It is a huge missed opportunity.

Below is an example of Go's dataframe library ReadCSV API:



```
func clamp(v float32) uint32 {
    if v < 0 {
        return 0
    } else if v >
        return mat
    }
    _ = dataframe.ReadCSV
    return uint32(v + 0.5)
```

```
func dataframe.ReadCSV(r io.Reader, options ...dataframe.LoadOption) dataframe.DataFrame
ReadCSV reads a CSV file from a io.Reader and builds a DataFrame with the resulting records.
dataframe.ReadCSV on pkg.go.dev
```

A few things to note:

- The API is cleaner and has the same functionality as the pandas one

- The documentation is not "too much", as was the case with pandas. It's clear what arguments are received and what is returned.
- There is a link provided by intellisense which takes the user to the online documentation where there's an example on how to use ReadCSV.

Final notes: Even in the case of a function with a simple function signature like str.join and file.writelines(), these receive a Iterable[str] and/or Iterable[LiteralString] and return either a str or LiteralString, which may be confusing for a first year student.

Intellisense is weak (Dynamic typing)

A type in Python can be unresolved which leads the IDE to show the type as **any**. In fact, this is more often the case than not. The following cases yield undefined types:

- Access to list, tuple or dictionary with no subtype (impossible to cover all cases)
- Arguments in a function when no type hinting is used
- Return value of a function when no type hinting is used

When working with undefined types all bets on what can be done with the type are off (as interpreted by the IDE). This may sound innocuous, after all people use Python every day with undefined types and no one bats an eye. The truth is it does not make programming in Python impossible. Most Pythonistas do not see this as an issue… that is until they try a statically typed language. The benefits of switching to a statically typed language are so numerous that this document is littered with the harms of dynamic typing all around. The author lists the benefits of static typing regarding intellisense:

- Ctrl+Spacebar (VSCode) is a powerful tool. It shows exactly **all** of the fields and methods that may be called on the type for **every** variable.
- It shows documentation for the type if documented.
- If the package system is strong enough it may also provide links to online documentation and examples of usage (e.g. Go)

Thanks to powerful intellisense teachers can begin to depend on it to teach. I.e. If a student is calling "split" on a variable of list type the teacher may tell them "Consider checking the methods available for that list type by clicking Ctrl+Spacebar" instead of the hand-waving involved with Python trying to explain where the variable came from to then have to ask the student to bring a function cheat sheet or to look for documentation online for the type.

Library cognitive overload

The packages most used in Python courses are numpy, matplotlib and pandas. It is the author's opinion these packages are far too removed in abstraction and complexity from what students can do before using them, especially in the context of a student who is new to programming and has never had to work with tabular, structured data. In the author's experience, students are barely grasping the idea of dict of dict records by the end of the semester when a DataFrame is presented to them. The motivation for this is that they'll be able to perform "SELECT * FROM X WHERE …" like queries in an idiomatic fashion using the boolean indexing scheme x[x>50].

There is no denying these are excellent constructs for easing the usability of Python. This has given the data science field unprecedented accessibility for budding programmers and an overall increase in the quality of life of data scientists. That said, is teaching budding programmers this concept favoring them? It is true this will make the students life easier for solving a subset of problems dealing with tabular data but we may have inadvertently doubled the difficulty of the course since we have introduced novel concepts:

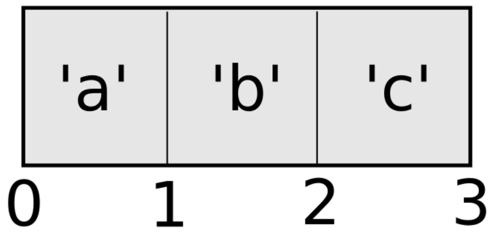
- **Extreme vectorization:** Yes, you can write a for loop to count the characters in a string but there is also the count() method! This is one of the early realizations of modularity- functions can embed for loops and save up on the code complexity (though in Python it is common to teach functions after loops due to the difficulty functions present with respect to types, see **Loss of context in code**). With the introduction of numpy's \`where\` or panda's \`DataFrame\` students are introduced to new ways to save up on a for loop and new ways to access data like DataFrame's iloc and loc, numpy's matrix access notation, DataFrames and numpyArray comparison operator overloading. This arguably introduces a DSL the students must learn to work with either numpy or pandas before they have assimilated basic/core programming concepts. It also considerably weakens the perceived power of the for loop, which may not necessarily be a bad thing but is worth pointing out.
- **Opaque/complicated Objects**: Whereas before teachers could work through a problem and print out intermediate results, now most values students must work with are opaque or require hand-waving explanations: i.e. the result of np.where is a tuple of two numpy arrays, the first containing the row indices and the second containing the columns indices.
- **Complex and obfuscated documentation**: Although Python is well documented the presentation of said documentation or in how its API has been designed could be better (See **Complex APIs**). Let's be forgiving and say Python's is "OK" in that aspect. In that case, pandas and numpy are "Questionable" at best since it would be ludicrous to suggest students depend on their documentation shown in intellisense or in online reference manuals while programming. Standard practice is to google what you want to do and visit a few Stack Overflow links.

Inconsistency

It may come as a surprise to some that there is a reason Python does not include the “end” element of a slice operation. In 1968 a committee was gathered in Paris with the brightest minds in computer science. Names like Djikstra, Naur, Wirth and Hoare were present. The result? Algol 68.

Algol 68 introduced slicing as we know it today in Python. It was decided that indices do not correspond to the elements as is common in Fortran style languages, but rather to the starting position in memory. An array could be represented graphically as follows

```
> slice = ['a',   'b' ,  'c']
```



So 0 corresponds to the start of the array, not the first element. When we ask Python for the contents between indices 1 and 2 we should get “b”. Sure enough

```
> slice[1:2]
> [‘b’]
```

For anyone learning slicing for the first time, the reason Python does not include the "end" element should now become apparent and intuitive.

But what if we want to slice it in reverse?

```
> slice[2:1:-1] # From position 2 to position 1, should be 'b' only following previous logic
> ['c']
```

Oh dear… well at least it does the expected thing for negative indices… right? Well yes, but only partially, you can't index from 0 towards the negatives.

```
> slice[0:-1]
> ['a', 'b']
> slice[0:-2:-1]
> []
```

Instead of following the memory model on which itself and countless other programs are based on, Python has chosen to invent its own memory model. The author suspects this is due to how Python is taught to most undergraduates with the aberrant saying: "slicing does not include the end element", thus, when reverse slicing was added to the language it preferred to implement it as it had been taught. Due to this the answer that could be given by Pythonistas when asked

Q : "Why does Python not include the last element?" is
A1: "Because that's how the computer thinks" or
A2: "That's how Python works under the hood" or
A3: "Because that's how it's always been done"

Of which the first two are false. A programming language is designed to benefit the programmer, not the computer. This is especially true of higher level languages and more so in education. If this were not the case we'd be writing our programs in assembly.

Due to this inconsistency in the language, it is dangerous to teach indexing as it exists in the world since this would mean students slip up in the case of reverse slicing. There are other cases of inconsistency including (but certainly not limited to):

- PEP encourages underscored function names, but standard library does not comply.
- str.isalnum(): You need to look at documentation to know what isalnum does. Why would they not use a self-documenting name for a method? Why does the Python standard library act like Pythonistas have to memorize methods (supposing IDE has the most basic type of intellisense) to be able to make the most of Python? This design decision violates at least 2 aphorisms in the Zen of Python.

Gotchas

As a teacher of beginner level Python, it's often the case that the need to teach lots of small details to students arises due to the creative design of the language:

- Not all whitespace is equal, even when they look identical in the IDE
- Python has "global" variables (module level variables) and **global** (when using keyword) variables. A module-level variable is "read-only" when it's not a primitive type (string, int, float). Dicts and lists are modifiable from any scope and thus not truly local variables. This confuses students who are taught that functions create a copy for arguments and variables outside of their scope. It also raises the question of pointers when in reality one of python's attractive features is the absence of pointers from the language's syntax model.
    - Problem arises when students use += operator with globals in a `def`
- Functions with no return return None successfully
- Tuple unpacking rules
- Looping: `for`, `while`, `enumerate`, `range`, `zip`, etc. Just so many ways to do the same thing.

Python harms ability to reason about a problem

“It's very illuminating to think about the fact that some –at most four hundred– years ago, professors at European universities would tell the brilliant students that if they were very diligent, it was not impossible to learn how to do long division. You see, the poor guys had to do it in Roman numerals. Now, here you see in a nutshell what a difference there is in a good and bad notation." **-Edsger Djikstra, 1977**

It is no secret Python is harmful when used to solve large scale problems. Today Python is being chosen less and less as a software engineering tool, only being preferred when the problem has a fixed scope. This is largely because of anti-features which are undesirable in a software engineering setting such as dynamic types, poor stdlib (standard practice not to use it), poor default tooling (pip), questionable community practices, insecure packaging system.

That said, what does "harms ability to reason about a problem" mean in this context? Consider these following quotes on the *importance of data structures in programming*:

"I'm a huge proponent of designing your code around the data, rather than the other way around, and I think it's one of the reasons git has been fairly successful… I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships." **-Linus Torvalds link**

"Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming." **-Rob Pike link**

"Much more often, strategic breakthroughs will come from redoing the representation of the data or tables. Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious." --**Fred Brooks link**

"The purpose of all programs and all parts of those programs is to transform data from one form to another [...] If you don't understand the data, you don't understand the problem [...] Conversely you can understand the problem by understanding the data." **-Mike Acton link**

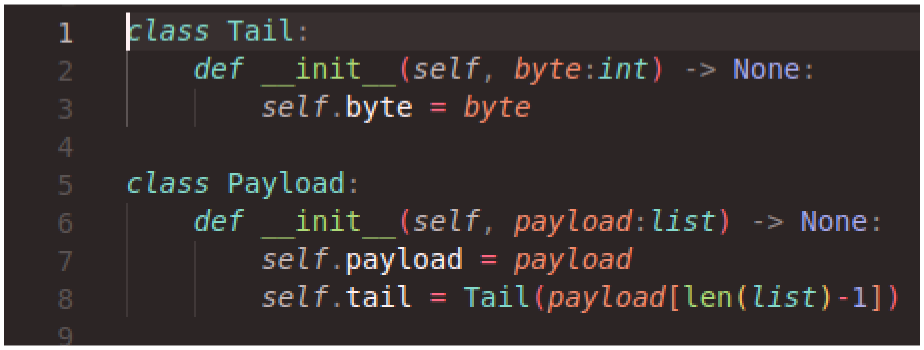
It would seem data structures are important when programming. The author claims one begins to reason about a problem by thinking about the data: what shape should it have to solve the problem at hand? Can this shape be generated easily? Where is the real cost of solving the problem?

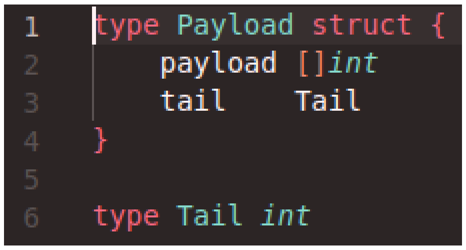
Python is limited to dict, list and tuple composite data structures in an introductory course. In more advanced courses Pythonistas may use classes. The next few points will make reference to the available data structures in Python as one of its core flaws.

Python's interpretation of OOP obscures data

Python's take on OOP is one that follows a tradition that "marries data with operations" [9]. The most glaring problem is not a fault in Python's features, but how Python encourages programmers to think about a problem. Classes put special emphasis on methods and little emphasis on its data. A class cannot exist without it's __init__ method, the data members only exist within the classes methods (as is apparent to an observer which inspects the class declaration). A long time Pythonista will learn to inspect the __init__ method or simply scroll down intellisense suggestions to learn what data members are available, though this becomes tiresome once classes begin to be embedded within classes. Composition is a pain point in Python.

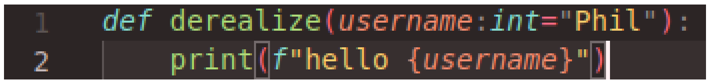
Consider the following


```python
class Tail:
    def __init__(self, byte:int) -> None:
        self.byte = byte

class Payload:
    def __init__(self, payload:list) -> None:
        self.payload = payload
        self.tail = Tail(payload[len(list)-1])
```
There is no semblance of data structure definition here. Only class boilerplate code which mixes with the initialization logic. Compare with a data-oriented definition of these data structures.


```go
type Payload struct {
    payload []int
    tail    Tail
}

type Tail int
```
The data relationship is clear, readable. There is no blurred line between logic and data structure. In the long run this simple change in data definition makes the code more readable and overall easier to work with since data structures are the core notion at the center of programming.

Dynamic typing

Python is a dynamically typed language. This is a boon for anyone writing short and sweet scripts to solve small problems. For larger more complex programs Python helps out by adding type hinting, which still does not fully solve the dynamic typing problem of python.


```python
def derealize(username:int="Phil"):
    print(f"hello {username}")
```
It may be of interest to the reader to know most if not all recent successful modern languages are statically typed:

- Zig
- Go
- Rust
- Carbon (not ready for use)
- Nim
- Swift
- Odin
- Kotlin
- Typescript (textbook case study)

This is because static typing acts as a guard rail. It avoids a whole subset of possible errors and crashes that your program would otherwise be vulnerable to. It also is often cited as the reason programmers are so productive when switching from a dynamically typed language to a statically typed language.

"We underestimated how powerful the editor integration is. Typescript was a boon to our stability and sanity". Felix Rieseberg on Slack's switch from javascript to typescript (as a testament to the benefits of a statically typed vs. dynamically typed language).

No user types outside classes

The author will tell of an experience teaching to illustrate the damage done by not having easily defined user types.

We were solving a problem that required reading data from a .csv file and inserting the rows into a dictionary as a dictionary, i.e. dictionary[id] = {"Name": name, "Lastname": lastname …} When we got to this point students froze dead in their tracks. The author suspects it was due to the cognitive overload of having to define a data structure while programming the algorithm. Most wanted to save the row as a list inside the dictionary since a dictionary inside a dictionary was "too much". This is what Python teachers have to wrestle with: **the cognitive overload of defining data structures while designing algorithms.** While there is a @dataclass decorator for defining classes with no __init__ method this still does not strictly prevent the intermingling of data and logic and provides additional cognitive load of the decorator pattern in Python. Is it reasonable that a package be imported every time a deficiency in Python's type system needs to be addressed?

```
from dataclasses import dataclass

@dataclass
class asd:
   fullname:str
   def firstname(self):
       return self.fullname.split()[0]
   def lastname(self):
       return self.fullname.split()[1] if self.haslastname else ""
   haslastname:bool
```

Consider a statically typed language. Before even starting the problem we may begin to think about the data structure for an employee by looking at the csv file. We reason about the .csv file contents first before writing a single line of code. We know we must define a data structure with fields corresponding to the amount of columns in the CSV. When we get to the point of saving data in our dictionary we now simply assign the defined data structure to the dictionary entry, much like filling out a template. There is no reasoning to do, it has already been done in the data structure definition stage. See "**Loss of context in code"** section for more information on struct types and fields.

Reliance on dict type
The lack of static types puts Python dict's type in an awkward position: It is by far the most flexible and natural feeling type in the language. Whenever there are "fields" in play, as is so common in data manipulation, the **dict** type is the obvious choice. It's hard to justify the use of a tuple or list to store fields when there are several fields in play since indexing by strings is just so much more readable than using hard coded number indices.
That said, when using dictionaries to store dicts within dicts the indexing can become confusing when stringing several indexing operations to reach a nested dictionary. Errors can become hard to follow and students must depend on their own wit to debug. This is largely solved by static types as mentioned before where not only the compiler helps, but more importantly there is intellisense to let you know what fields are available and where. The sanity this would provide to students cannot be overstated. See "**Intellisense is weak"** section.

Only one way of doing things

It's in the Zen of Python, so surely Python would adhere to it?

- str.format vs f-strings
- list.reverse() vs. list[::-1]
- list[:] vs. list.copy()
- Iterations
    - While vs. for. Having two distinct keywords for doing the same thing (iterating). Keywords add things a student new to programming must memorize and hampers the cognitive process. Citation pending.
    - For vs. list comprehension
- The list goes on

Note: The author is not suggesting str.format be removed from the language, as this would break backward compatibility. This is merely an argument against using Python in education.

The concept of orthogonality (in context of programming languages) has been cited as a boon to the ease of learning a language and the productivity it brings [6]. This is mainly because novice programmers may get caught up with the amount of tools at their disposal and instead of focusing on solving the problem, they focus on choosing the most adept tool for the problem.

"Programs have to be fast"

Throughout his career the author has seen his Python programming colleagues have a misguided obsession with program execution speed. More often than not, these colleagues who teach Python to undergraduates want to show students big-O notation by the second class.

The author's perspective on this matter is as follows: *Reason about the problem and build the simplest thing that works*, **only optimize later if:**

- The program's execution speed is a problem
- You can **measure** where the program is slow (profiling). More often than not your guesses will be wrong on where the bottleneck is [4]

The author is not sure where this obsession has come from but it seems it is part of a wider more systematic problem in the Python community, often manifesting as a command line flag such as `"--fast"` which results in faster programs with less correct results. Examples of this in action:

- Widespread package poisoning with FPU flags
- Black (--fast)
- this developer who'd rather matrix multiplications yield completely incorrect results over being marginally slower.

In any case, by using Python you are already coming at a loss in terms of performance, not only because of the underlying implementation, but because Python impedes reasoning about data structures (talked about earlier). Data structures and algorithms are equally important in defining program execution speed and far outweigh compiler optimizations and program tuning [4].

Other

Python prevents innovation

Python limits teachers when they want to create new and interesting problems for students to solve. Python does this a few ways

- Poor standard library: Usually need a specific library to solve a problem, i.e. http requests, image manipulation etc.
- Poor packaging system: Installing libraries is a game of dice, maybe the library does not play nice with the local python environment. Thus anaconda and other packaging systems are widespread
- Wheels availability and correctness: Maybe the python version on the students machine does not work with the library. Preparing wheels is also error prone work, the author is familiar with errors that appear on certain architectures on certain OSes.

Thus, when a teacher wants to propose a new problem they have to jump through a few hoops: Have the students installed a library that helps them solve the problem? What libraries do the students have to install to solve the problem? What issues may arise regarding versions of the wheel, library semver, the student's install environment.

Due to these issues it's often easier to just settle for a set of libraries when the semester starts. This means teachers have to iterate course work on the order of months regarding new problems. This is especially a problem with new courses that do not have tried and tested course material. Most modern languages have solved this issue in full. Take for example Go modules:

- No manual library installation needed. Ever.
- Robust versioning: No cyclic dependencies, minimum version selection, reproducible build file.

Due to this above, one can run pretty much any .go file and the go tool will take care of resolving the dependency graph and installing packages. So in practice a teacher can give students a .go file and a student with the same Go version will be able to run it on their computer problem-free, thus a **teacher may formulate any problem using any library**. What's more is that publishing a package is as simple as creating a repository with 2 files on github (.go and .mod files). This is because go packages are URLs. A **teacher can then publish a library overnight so that students may use it the next day**. This is a game changer for innovation in coursework.

Loss of context in code

Context helps the programmer deduce what the code is doing. This can refer to

- Identifiers: a.k.a. Naming. What data does a variable contain? What does a function do?
- Types: Give form to our data. How does it contain the data it contains?
- Syntax annotation: Python is an imperative language. There is syntax to the language, how does the syntax help convey what we want the computer to do? This can come in the form of keyword combinations, operators, and formatting of the code (indentation, operator grouping, …).

Take the following Python function

```python
def update_salary(employees, salaries):
   for id, employee in employees.items():
       xp_employee= employee["Experience"]
       for xp_minimum,salary in salaries:
           if xp_employee<=xp_minimum:
               employees[id]["Salary"] = salary

   return employees
```

Let's break it down

- We may guess employees is a dictionary after seeing the items method call.
- Employee seems to contain dictionaries as it's values (from the "Experience" key access)
- Salaries looks like it could be a list of tuples. The tuple in it contains xp_minimum and salary
- We know nothing of these types except that xp values may be compared to set the employee's salary when a certain threshold is met (xp_employee <= xp_minimum)
- The whole function returns the modified employees data structure

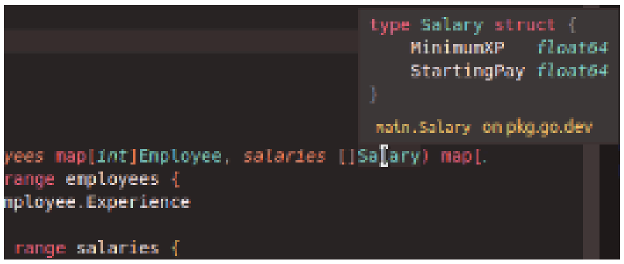
Let's see a statically typed language implementation of the function

```
func update_salary(employees map[int]Employee, salaries []Salary)
        map[int]Employee {
    for id, employee := range employees {
        xp_employee := employee.Experience
        for _, salary := range salaries {
            if xp_employee <= salary.MinimumXP {
                employee.Pay = salary.StartingPay
                employees[id] = employee
            }
        }
    }
    return employees
}
```
A few things we did not know about the Python implementation (or that had us guessing) become immediately apparent:

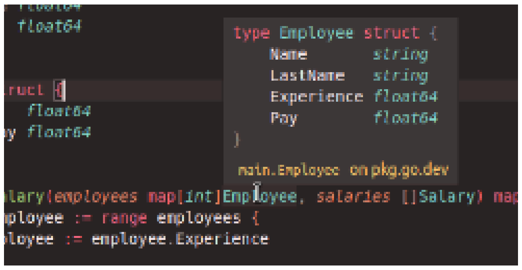
- We must access employees dictionary using an integer
- Salaries is indeed a list containing the type Salary!
- We immediately know the function returns a single value and its type. We get a much better sense of the work this function does at a glance!

We can then mouse over the Salary and Employee types to acquire the full context experience:

- Salaries is a list containing a Salary with the fields MinimumXP and StartingPay
- Employee contains various fields we had no knowledge about such as Name and LastName



```
type Salary struct {
    MinimumXP   float64
    StartingPay float64
}
main.Salary on pkg.go.dev
yees map[int]Employee, salaries []Salary) map[
range employees {
mployee.Experience

 range salaries {
```



```
float64

ruct {
  float64
y float64
                 type Employee struct {
                     Name       string
                     LastName   string
                     Experience float64
                     Pay        float64
                 }
                 main.Employee on pkg.go.dev
lary(employees map[int]Employee, salaries []Salary) ma
ployee := range employees {
loyee := employee.Experience
```

We now have a much better idea of what the algorithm does. We had no idea that the numerical salary inside a Salary type was the starting pay and we know the experience inside the salary type is the minimum experience needed to be able to get paid said StartingPay. We managed to figure all of this out without a single code comment. This is the power of having local code context with types and intellisense. This is all lost when using a dynamically typed language.

Python's carbon footprint

Python is one of the most inefficient languages in terms of energy consumption one can use. It consumes 70 times more energy than C and an order of magnitude more than similar languages. This means Python's carbon footprint per user is larger than that of other languages when using pure Python.

Why is Python like this?

This subsection aims to answer a question the reader may or may not have: "Why is Python the way it is with respect to these problems?" by contrasting the history of the design of Python with the design of Go, a language that does not have the deficiencies outlined.

Python

A few things to note about Guido Van Rossum and his creation, Python See [Oral History of Python Part 1]:

- Bright student, among top of his class.
- Knew Pascal, Algol, C and Shell before designing Python
- Worked on the team that created ABC, a scientific language and spiritual precursor to Python
- Python was born out of frustration with C and Shell. Van Rossum had to write programs that "took a week to write in C, but that could be written in 15 minutes with ABC"
- Van Rossum designed language on their own, taking inputs from colleagues once the groundwork was laid down
- Van Rossum had the final say on what went into the language for the next ~20 years

Go

Go is a language similar to Python, often touted as "the next Python" due to its striking similarity and simplicity. The language was conceived by Rob Pike, Ken Thompson, and Robert Griesemer. Some notes to contrast with Python: (see [Building on the Shoulders of Giants, by Steve Francia]

- The amount of language design experience these three had collectively is mind boggling. Ken is the creator of the C programming language among others
- Rob Pike's papers are often cited by language designers as being core to language design decisions (see Odin)
- Go was born out of frustration with C++, Rob Pike has often cited feature overload and readability as the problems with C++
- The three were working on a Google software project that spanned millions of lines of code
- The three had to be talked into a feature before being added to the language
- Go was open sourced upon its release and follows a public driven change policy

The author hopes that the points above offer some insight into why Python has the problems outlined in this document.

It is the author's **opinion** that Python had little input from others throughout the design and was purposefully designed to solve small, bite sized problems that have a very low cognitive load when being read or solved.

Solutions

Go

The author has come to the realization that Python is unfit for educational purposes because he teaches Python being an experienced Go developer. He has hit the walls mentioned above and reminisced of how it would be different if teaching Go. Truth be told, **all of the issues mentioned above would be solved by switching from Python to Go** as the de facto educational language. Benefits/disadvantages of Go:

- First look at Go is not progressive. There is cognitive overload when seeing three keyword declarations in a file just to print "Hello, World"
- Intellisense breaks when storing multiple programs in a single folder. Must keep a single Go program per directory. This may confuse first time users and users coming from Python
- Lack of built-in operators such as **in.** Instead can use slices.Contains
- No stringing comparisons. I.e: a < b < c
- No need for manual package installation. No virtualenvs either
- Quality stdlib. Image manipulation, http server and requests. Students can solve more interesting problems thus helping motivation
- One way to do things, really. Orthogonality was something the designers had in mind.
- Students will have to depend on standard library which will expose them to packaging and namespaces from the start
- Students will have to write their code inside the main function which will expose them to function syntax from the start
- Autoformatting is forced. All Go code in the world is self-similar
- No-nonsense functions. No *args, **kwargs, keyword arguments (when abused can be noxious, especially when combined with the likes of PEP570 and PEP3102), global keyword, decorators.
- Go's for loop is confusing in a different way than Python's iterators. Go has only one keyword for iterating "for", though the for loop can be written in roughly 2 (three if you're strict) ways. The advantage of the for loop in Go is that it can be taught incrementally. This is important as it is the hardest topic students in the course see (in the author's experience).
- Progressive loop difficulty: In the author's experience, one of the hardest topics for students has been the while loop. Go's while loop does not have many of the pitfalls of Python's while loop
    - Can copy paste code into Go's while loop without risking indentation errors
    - Impossible to mismatch whitespace
    - Simpler syntax: **for { fmt.Println("Hello") }** is a valid while loop in go, can be taught with minimal new syntax (just a new keyword, no statement). Reduces cognitive load and makes it overall easier on students.

There are of course disadvantages and limitations to using Go for education when compared with Python.

Kotlin

Same playing field as Go. Would solve many of the aforementioned problems found with Go. Not as popular though and therefore may be harder to find online documentation, help, libraries.

- Heavier syntax than Go
- Struct types are slightly more obfuscated than Go

Swift

Similar playing field as Go. Solves the issues with Python. Is actually marketed as a good first language by Apple. Swift+Xcode is being taught at Shanghai Business school to focus on developing apps. A few notes on swift:

- Much heavier syntax than Kotlin/Go
- Relies on exceptions which are hard to teach since it is a form of systems programming
- Is arguably more complex than Kotlin/Go

Solutions Continued: Honorable mentions

Hedy

This language finds the global optimum way of teaching Python.
https://www.youtube.com/watch?v=fmF7HpU_-9k&ab_channel=StrangeLoopConference

- Progressive language: levels means concepts are taught little by little, ensuring minimum cognitive overload
  - Great for middleschoolers
- Built for teaching: Has a curated coursework program
- Still Python. Has the same problems outlined above.

Zig

Much closer to C than Python, though it is a great language to design low level systems and solves many issues explained here. Best suited for engineering/compsci students. Still in early development.

Dart

While it would solve proposed problems with Python it seems like a cluttered language at a glance since it is optimized for developing GUI clients.

Rust

The most promising language in the world is thought to be too cluttered and low level to teach as a first language. Best suited for advanced engineering and compsci students.

Julia

The author has heard of Julia being used in education for math and physics courses. However the author is wary of Julia since it is a dynamically typed language (back to square one) and has been known to be faulty and unfit to build robust software (article link). The Julia authors are known to be overburdened with requests for fixes.

Go+

Basically Python but marginally better. Although it is statically typed it loses readability since types are not compulsory. Still presents some of the problems presented in this document.

Sources

The author of this treatise has taken the liberty of adding bold typeface to passages of interest.

[1] Teaching 11 year olds to program

From https://groups.google.com/g/golang-nuts/c/FIRSDBehb3g/m/BFiHYVNCwzUJ?pli=1

First message

I'd like to describe my experiences using Go as a first programming language for a group of young programmers. [...]
This is the first question I have seen that directly relates to teaching children.
My situation was similar to Maarten's. I had 12 **eleven-year-old school children**. Over the course of the last 7 weeks of the school year, about 12-13 hours in total, I managed to **teach them just enough Go so they could write a mandelbrot generator.** But,critically, they were able to understand the code.[1]

And what I found was: Go is not only a good teaching language, it's an excellent one for first time programmers, including children. I want to try and outline the Go features that really stand out when you have young programmers and why I think we've ended up here.

The first feature that helps is Go's left to right syntax. If you are eleven this intuitively makes sense because Go reads in a natural way. This becomes very apparent if you ask the children what they think a particular line means.

Secondly, Go has a small set of keywords. If you are child this turns out to be important because it seems like there's not a lot to learn. They can (initially) re-frame the problem of learning to program into "What do these words mean, and how do I use them?" Now the problem looks to be tractable to them.

Thirdly, go fmt is a huge help on a number of levels. It's a confidence boost to a child if they know that they can just type a program in and not worry about the exact formatting, knowing that the editor, via go fmt, will fix that for them. Over time they learn what the go fmt style is and just start doing this naturally.

Go fmt by its nature makes everyone's code look the same; this has an interesting side effect. When (not if) they spontaneously start helping each other, and they start comparing a program that works to one that does not, they are not looking at the formatting, they are focused on the logic. They actually compare the order of the steps in each program to find the differences and fix the problems. Go fmt shortens the mental leap you need to do this, without it this process may not have arisen as quickly as it did and would have required a much larger mental leap to see though the formatting differences. So go fmt as it turns out is actually an aid to learning and understanding. It lowers the barriers a child needs to understand a program. My conclusion from this is that eleven-year-olds need go fmt for the same reasons we do.

Fourthly, the go tool and the workspace. The go tool makes things very easy for young programmers to get started. Once they learn that all they need to do is use 'go run' to run their program they never ask how to run any program again. I only had to show them this two or three times.I just cannot imagine doing this with either makefiles or a string of command line switches. This is so simple that children just get it.

The workspace also helps. Simply knowing that they have to put their code under $GOPATH/src for it to work helps because it forces everyone to do the same thing. The children don't have to worry about the program not building because it (or a dependency)is in the wrong place.

When I started this, I thought that the children would stand a good chance of being able to use Go. But there were a few areas that I thought might prove problematic when I tried to explain them to the children.

Types are an interesting case. I thought this might be a really a hard concept for the children to grasp because it is fairly abstract. Most of the usual teaching languages used with children are "typeless"for this reason. **But exactly the opposite was true.** The children just got it. They only had to make the mistake of trying to assign an int to a string once or twice to realise that the compiler won't let them do this. The compilers static checking really helps here, because it stops the children and tells them there's a problem *here*.

But of course, having types also helped.The children had to reason about them when they first declared the variable. They would talk about what they wanted a variable for and then pick the type they needed. In a very subtle way this extended their logical thinking abilities.

I used Atom and a command line to teach the children, rather than an IDE. I was concerned that the lack of a UI and "Run button" might be a problem for the children. But they proved me wrong. Provided they have a syntax colouring editor, with go fmt integration, and they are shown what commands they need to build/run their program it's not an issue. Using the command line really wasn't a problem. Similarly they didn't need a debugger. When their programs went wrong they just went back to the editor changed it, rebuilt it and tried it again. The edit/build/run cycle is so quick they just didn't need a debugger. If anything having to use a debugger would have slowed them down. But this may, in part, be due to the smaller size of the projects they were creating.

Marking blocks with braces also wasn't a problem for the children. The children just never questioned it. Perhaps partly because go fmt also sets to the indentation to match. Or perhaps it's because they have never programmed before they had no preconceptions about the presence or absence of braces. At the minute its not clear to me which case is true for the children. When they did miss a brace they soon learned to decipher the compiler error message a look to see where they had missed a brace or two.

What do I conclude from all of this? I don't think the Go team ever intended Go the be a good teaching language, but by a happy accident we seem to have both a good system programming and a good teaching language. A rare feat indeed. The only thing I can put this down to is the languages design process itself. Rejecting more than was kept and above all keeping things simple and orthogonal - both the language and the tools - has paid off in a way we might not had predicted.

As both Russ Cox and Andrew Gerrand pointed out at GopherCon this year we as a community must not lose sight of these original goals as the language moves forward. If we do we might be risking the programmers of tomorrow as well as today!

Lastly if anyone else has tried teaching Go to first time programmer, especially children, I'd really like to hear what your experience has been.

I already have plans to teach another larger group over a longer time period from September.

Regards
Owen

Second Message

Hello Everyone,

Okay so as a few people have asked "How did I do it?" I'll try and explain....Apologises if this gets a little off topic for the list.

Essentially I used a pattern based bottom up approach, with each stage building on the last one.

So we started with Hello World. Not for hello world itself, though they get a sense of achievement from printing this, but to show them how to edit/build/run a program.

Next we looked a numbers, so ints (*) to do simple sums (+. -. * and / operations) with small'ish numbers (<1K).

Next we looked at strings, with a simple program to print their name and age.

I think it is important to say that at this point there where no variables in the programs. Everything was static. I was just trying to get them used to typing code and beginning to understand little bits of it. So showing them how to print with fmt.Print and fmt.Println and reinforcing the edit/build/run cycle is more important at this stage. At this stage I'd showed them the pattern to print to the terminal.

Then we did variables, for both numbers and strings. I used a pattern approach to teach them this. So they had three patterns. Declarations in the form of "var variable-name variable-type", assignment in the form "variable-name = variable-value" and then the usage which is just the variable name. And yes, everything is long hand - there is no := operator. You can't use that until you understand what you are short-cutting. Also you want to make the types explicit so that they think about these. Kids with good maths skills will quickly see that variables are like unknowns in maths. Others you need to take a little more time with, before they see that the computer will substitute the variable value, when they use the variable name. I introduced this with a version of the strings program that used variables to hold their name and age.

Then we looked keyboard input so we can set the value of the variables at runtime. For this I wrote a simple wrapper around go's stdin stream handling. fmt.Scanf is fine in the stdlib, but it'll behave in an unexpected way if you feed it invalid input i.e. strings when its expecting ints etc. The behaviour is correct, but its not very encouraging if you are a child. The alternative I used was to wrap a read buffer around the stdin stream in a function (in a new package) without showing them the internals. That gave them a simple "Read{Number, String}FromKeyboard" function they could just use. To explain how that function worked I'd have to introduce interfaces, pointers, pointer receivers and streams which isn't appropriate at this stage. I wanted then to focus on using input to set variables. Not worry about how the input magic worked.

Then they did if and if-else statements again with patterns. So the if pattern became "if condition { true-statement-block }". I deliberately didn't show them the initializer block form to keep things simple. At this point they can start to write something "useful" so I got them to write a simple "quiz" that picked two random numbers, a and b, (between 1 and 12) and asked them to type in the answer to a * b. Then print out congrats, or bad luck depending on their answer. I had to take time with conditions, to be clear that the answer can only be true or false. So sometimes you need encourage them to "rephrase" the question. Also "==" took a little while to settle in, just because they haven't seen it before.

The last area they looked at was loops in the simplest form of "for condition { loop-body }". They used this to extend the previous program so that the program asked them a different question each time until they got the correct answer.

Once you have variables, if tests and loops you have enough knowledge to draw a mandelbrot plot. You only need 3 loops and an if test to do it. So I wrote a skeleton program that used the go SDL bindings to open the window and do the graphics parts, but left the calculation bits out. I needed to show them a little bit about screen coordinates (origin is top left, Y axis is down etc) vs. set coordinates so they can work out the scaling calculations. The pupils then had the follow the comments I left in the program to do the calculation.

I know this might whole approach might sound overly simplistic. But to get an eleven year old to this point will really stretch their ability to logically reason and problem solve. Sometimes we as adults forget just how much we know and take for granted.

Also I'm not trying to teach them idomatic go at this stage, or every language feature. That misses the point I think. What I was trying to do was to encourage then, and spark their curiosity, and interest. You want to remove as many barriers as you can at this stage. Once they stop thinking about what an if test does a how to write a loop or declare a variable you can start to building towards this.

Other more general tips if anyone else is trying this:

* Aim high, so pick something you think bright kids can do, then go a little bit further. Even I didn't think they would mange the mandelbrot plot when I started.
* Give them a goal, in this case they had the mandelbrot plot as goal from day one, that they can aim for to motivate them.
* Give them something fun or unusual, or something they ask about as the goal.
* Go slow, use little short lessons and build upon previous concepts.
* Go in a logical order. By this I mean don't aim for a http server until you can explain every concept that you need to use the stdlib code and have them understand as well. Start with the absolute basics and work upwards.
* Don't spoon feed them. By this I mean you can give then a complete program for the first two or three times. Once they get these working challenge them to extend them in some simple way. So print their friends name and age as well as theirs. Then as you go froward start to give them programs that are more and more incomplete and get them to fill in the blanks.

Owen

(*) I'm going to add floats into this in September as "float64" threw them when they saw it in the last lesson. The kept asking what the "64" meant.

[2] Using Go in the classroom

From https://groups.google.com/g/golang-nuts/c/ewJpIYNXSvs/m/oWQh9XCahdsJ
kev...@google.com
no leída,
23 dic 2012, 4:28:53
a Danny Gratzer,golang-nuts

[...]One thing that will really help the students is the fact that Go is **statically typed** and that the compiler is relatively strict. It will seem difficult to them, especially if they are coming from a language like Python where a mistyped variable name doesn't prevent the whole program from running, but in the long run I suspect that it will be catching a lot of their most common bugs before they even have a chance to materialize. The fact that Go is garbage collected, as I'm sure you've already surmised, is great for beginners because it reduces the amount of bookkeeping that they need to do. The lack of pointer arithmetic and the safety of the language are also great safety nets for a beginner.

If I were teaching a class in Go, I would probably start the students out with the usual hello world, followed almost immediately by the **hello world web app**. You can let the students play around with that in any number of ways, and it allows you to dive into a lot of important things that they are leveraging without having to code themselves. You can show them the standard library documentation, which will hopefully spur some of the students to go exploring. You get to explain about functions and function types. You get to explain about interfaces (the transition from http.HandleFunc to http.Handle with the same handler function is particularly interesting). When you've finished deconstructing that, you can move into more aspects of the standard library like templates, math, i/o, exec etc. Again, they'll be using a lot of features that you get to explain (types, fields, methods, etc) without having to actually reproduce it themselves. At this point I'd probably spend some time going through the language spec, so that the students are familiar with how it is laid out and what the features are called, so that as you start to ask them to create their own types, their own methods, their own interfaces they will know where to look for answers. As you do start asking them to create more of their own types and things, it probably makes a lot of sense to teach them unit testing. One of the requests I would usually ask a student who came to me for help with their code was "Show me your code the last time it was working, show me what you changed, and I will probably be able to tell you why that doesn't do what you thought." In retrospect, if we had instilled unit testing in them, it would probably be more like "Show me your code the last time your tests passed" or "Show me the unit test you wrote for the feature you're having trouble implementing." If you've made it this far with the students, I'd probably dive into concurrency patterns and maybe do some case studies with some standard library packages or third-party packages to have a look at real-world Go code and perhaps get some exposure to common idioms and documentation standards. More detail on networking and building client/server applications is another advanced topic.

[3] Courses that use Golang

[4] Perfbook

[5] Language Design in the Service of Software Engineering

[6] Testing the principle of orthogonality in language design

[7] Discord conversation with Tim Stiles and Mihai Todor

Tim, Author of Poly(merase) library:
I made a very conscious choice to use Go for this project based on several criteria.
Speed of development, speed of code execution, strong devops ecosystem, and being able to compile to a binary were all higher priority than what language other people were using in synbio (mostly python)

Go was the only language that fit all those criteria. Rust handles strings in a somewhat tricky and unique way that I thought would scare too many devs coming from python away, and its devops ecosystem wasn't really mature when I started the project.
I've seen some people actually posit that with the advent of Go generics that Go should become the default language for scientific computing for all of the reasons I chose it several years ago.

Mihai, Principal Software Engineer working on https://benthos.dev**:**
Back in 2015, I started working in a company where, just like in my previous 3-4 jobs, I was asked to contribute to a largeish C++ codebase that took well over 20 minutes to build (over 3 hours at a previous-previous job). I was so frustrated that, yet again, someone duped me into wasting my time with their horrible codebase and coding practices that I started talking about it with a colleague, Karl (https://relistan.com/), from the cloud infrastructure side of things. He told me to jump ship and learn Go, so I did. I told my management chain that they either assign me to Karl's team or I'm gone. After one year of mucking around with a few small internal projects, I was finding my way around Go codebases quite easily and I got a good sense of the various languages (Python, Terraform etc) and tooling (Docker, Ansible, Kubernetes etc) and systems that get used in this space. It was quite handy to have this guy as a mentor and see him code almost every day. /rant Here's what I think makes Go compelling: - Build speed and quick iteration cycles when running tests. You can do the same in Python, since no static compilation is needed, but the tooling and frameworks are not as snappy. - Parallel programming. Go makes this easy even for beginners and I don't think you can achieve the same thing in Python. I suppose there's some way to get Python to use a thread pool and then have some channel signalling mechanism on top, but it will require quite a bit of knowledge to get right. - Opinionated code style and small language footprint. No need to learn a ton of language features and quirks and many of the existing examples tend to be quite clean / well maintained. - Frameworks and libraries that are well-polished and idiomatic. The Go module tooling makes it

trivial to avoid versioning hell without virtual envs. - CGo. Calling C APIs is also well-supported and there's good tooling around it. - Gravity. It attracts people who like performance and got fed up with slow builds.
I think people will still pick Python and R over Go for many data science tasks, not just because of the gravity factor, but also because in Python they don't have to worry as much about types and curly brackets. I've seen https://goplus.org/, but it still doesn't feel like something that will get people to jump ship... Also, somebody showed me this https://dashbit.co/blog/nx-numerical-elixir-is-now-publicly-available recently, which I guess might be cool to try, given how much praise Elixir gets in some circles, but I'm not sure it's worth the time investment to learn it when one can achieve the same stuff using languages that they're more familiar with.

[8] Prat, Chantel S., et al. "Relating natural language aptitude to individual differences in learning programming languages." *Scientific reports* 10.1 (2020): 1-10.

[9] CppCon 2018: Stoyan Nikolov "OOP Is Dead, Long Live Data-oriented Design"