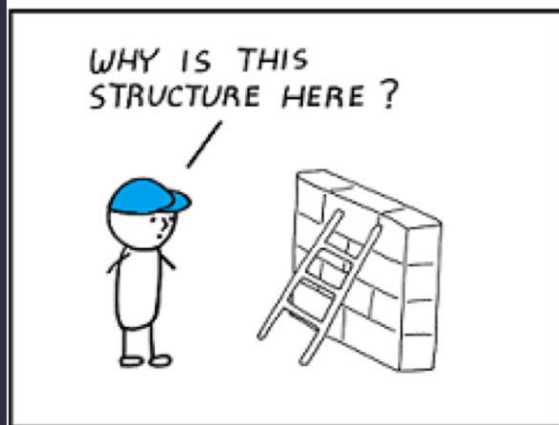
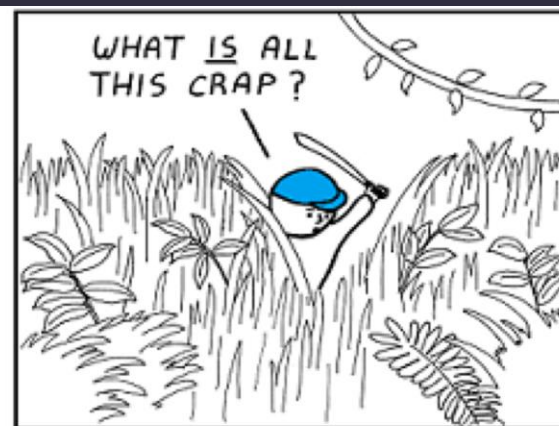


# Clear Code

---

WRITE CODE THAT IS EASY TO UNDERSTAND



# Why write clear code?

---

- ❑ Easy to understand
- ❑ Looks professional: inspires trust
  - Encourages others to not start from scratch!
- ❑ Makes bugs harder to hide
- ❑ Code clarity does not conflict with performance!
- ❑ Important for your collaborators!
- ❑ Clear code is easier to reproduce and retrace
- ❑ *Future you is a frequent collaborator...*
- ❑ Makes it easier to extend

# Naming: it's important!

---

- ❑ Which of the two versions better conveys what the function does?

```
function get_them(b)
  list = []
  for x in b
    if x.status == 4
      push!(list, x)
    end
  end
  return list
end
```

```
function flaggedcells(gameboard)
  flagged = []
  for cell in gameboard
    if cell.status == FLAGGED
      push!(flagged, cell)
    end
  end
  return flagged
end
```

- ❑ Code is identical and the operations are understandable in both cases
- ❑ But motivation, reasoning and outcome is much clearer in the right version!
- ❑ Naming is crucial for conveying the story of what the code does!

# Naming: brevity, or not?

- ❑ Use intention revealing names and avoid abstract/meaningless/1-char names (e.g., data, object, foo, baz, a, b, c...)
  - People find concrete examples easier to follow than abstract examples
  - There is always something specific that better suits your case instead of a generic name
  - *always err on the side of clarity instead of brevity*
- ❑ BUT! Brevity does have its place in...
  - “Dummy” variables which only have meaning within the code block they are defined (like i, j in the example)
  - Variables whose name is only meaningful in the caller, e.g., `mean(x)`. What is `x` is only known by whoever called the `mean` function. So the source code of `mean` can just use a simple name like `x`

```
a, b, c = 1, 2, 3
# versus
bird_speed = 3
days_since_creation = 1
```

```
for right_index in 1:10
    for left_index in 1:10
        m[left_index, right_index] = rand()
    end
end
# versus
for i in 1:10
    for j in 1:10
        m[i, j] = rand()
    end
end
```

# Naming: bad brevity example

---

- ❑ Besides being uninformative, brevity may also lead to too-similar names...
- ❑ Pop quiz: what do you think these variables represent? `rsdt` `rsut` `rsus`
  - `rsdt` = shortwave irradiance at the top of atmosphere, incoming
  - `rsut` = shortwave irradiance at the top of atmosphere, outgoing
  - `rsus` = shortwave irradiance at the surface, outgoing
- ❑ *Typing characters isn't a big deal! Use long descriptive names!*
- ❑ Too similar variable names may be silently confused or mistyped!
  - In fact, these names are so similar that it is statistically certain that you will mistype one for the other: their Levenshtein distance is only 1 (basis of autocorrect algorithms)

# Naming: Unicode

- ❑ New programming languages allow usage of Unicode characters *in code*

- ❑ Some advantages of Unicode:

- Math expressions feel natural:  $\psi_1 \otimes \psi_2$
- Exactly write a paper formula in code
- Define unique symbols that you use throughout your code base

```
cross(psi1, psi2) ==  $\psi_1 \otimes \psi_2$ 
```

```
 $\nabla \rho$  = gradient( $\rho$ )
```

```
 $\langle \varepsilon \star \rangle$  =  $\Gamma * \text{mean}(\varepsilon \star) / \lambda$ 
```

```
SW_TOA_↑ = upwards_solar_radiation(data)
```

```
SW_TOA_↓ = downwards_solar_radiation(data)
```

```
 $\alpha$  = SW_TOA_↑ / SW_TOA_↓ # surface albedo
```

```
const  $\Re$  = Real
```

- ❑ Some disadvantages of Unicode:

- Hard to input Unicode in some environments, and some terminals cannot display Unicode
- May be unclear what Unicode code to use, and some Unicode symbols look similar ( $\alpha$  vs a)
- Some Unicode symbols can be confused as operators (like the star here)

- ❑ Use Unicode for read-only things (docstrings, low-level code, comments), but avoid for writable things (function names/keywords, re-used variables)



SAFELY ENDANGERED



SWEET JESUS, POOH!  
THAT'S NOT HONEY



YOU'RE EATING  
UNICODE TEXT



□□□□□□ □□□□  
□□ □□□□ □□□□



# Naming: what to avoid

---

## ❑ Avoid misinformation!

- Don't name a variable `customer_vector` if it is not a vector, but e.g. a dictionary
- Don't name a variable `customer_dict` if it is the dictionary of employers

## ❑ *Avoid constant literals*

- E.g., don't write "2020", write ``year = "2020"`` and use ``year``
- Assign as many constant values to variable bindings as possible
- Makes your code more generic and extendable!

## ❑ Avoid numeric suffixes: (agent1, agent2), (pos\_1, pos\_2)

- Sometimes enumeration by integer is the most fitting, true, but...
- But most often than not it isn't! The variables will likely have more meaningful separation
- E.g.: (agent\_junior, agent\_senior), (position\_old, position\_new)

# Naming: what to avoid

- ❑ “Hungarian notation” where name indicates the type, e.g., `customer_dict`, `employers_vector`
  - *if* the type is obvious from context
  - *or if* your programming language is typed, and hence allows specifying the type programmatically within the language

*# `\_dict` is useless here:*

```
birthdays_dict = Dict{String, Date}()  
birthdays = Dict{String, Date}()
```

*# `\_vector` is better specified as type*

```
function employer_birthdays(employer_vector)  
    for employer in employer_vector  
        # ...
```

```
function employer_birthdays(employers::Vector)  
    for employer in employers  
        # ...
```

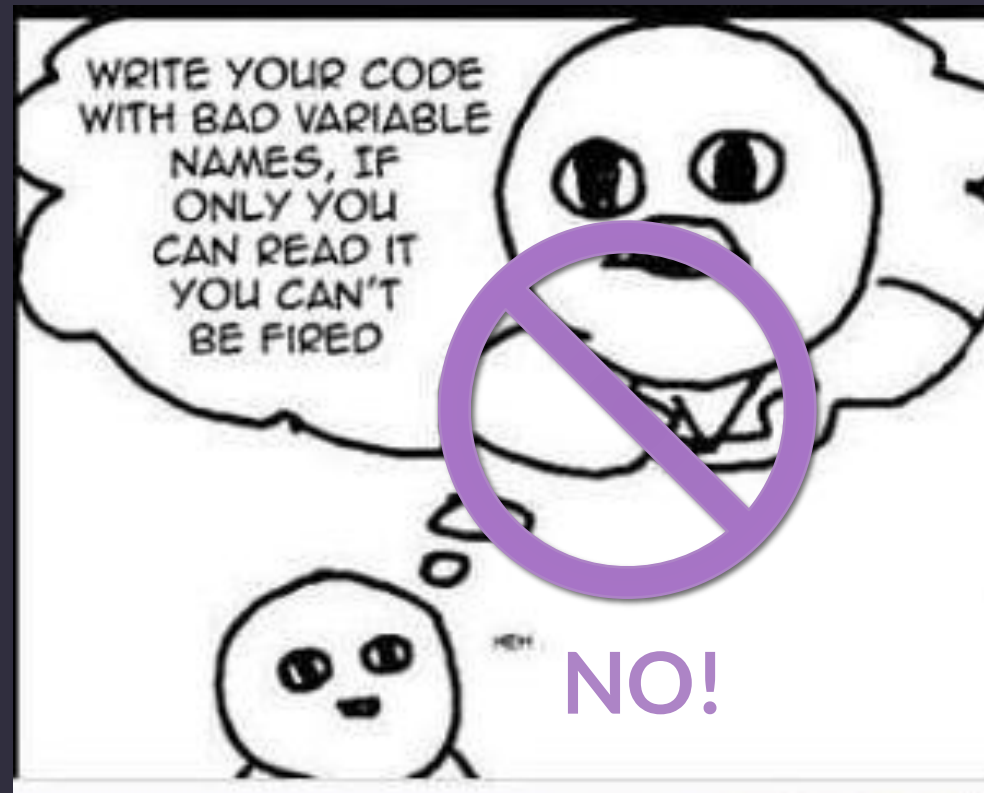
- ❑ Only exception is naming variables starting with ``is_`` or ``has_`` if they are Boolean

```
has_water = air_parcel.temperature < saturation_temperature(air_parcel)  
is_prime = factors(number) == [1, number]
```

# Naming: follow the language style

---

- ❑ In e.g., Julia or Python, the following convention for naming is used
- ❑ Functions and variables are lower case with possible underscores
  - `example`, `example_with_long_name`
- ❑ Types/Classes are camel case without underscores
  - `Example`, `ExampleWithLongName`
- ❑ Global constants are all capitals with possible underscores
  - `EXAMPLE`, `EXAMPLE_WITH_LONG_NAME`
- ❑ In Julia modules are CamelCase, in Python they are lowercase without \_
- ❑ Following conventions = super cool
  - Meets existing expectations and conveys role of variables without necessarily reading all code
  - Not following conventions will confuse user or make them uncomfortable
  - e.g., using all capitals for a function name or using snake case (`snakeCase`) in Julia



# Exercise: bad names

---

- ❑ We're providing the file `badnames.jl`. Try to figure out what the functions do.
  - Not so easy with bad names, huh?
- ❑ Re-write the names of variables/functions to make the code easier to navigate and understand. Do not alter the code operations in any way! Only the names!
- ❑ ***Max time: 10 minutes!***

# Functions: functional programming


---

- ❑ **Functional programming** is a paradigm particularly suited for scientific programming
  
- ❑ What is it?
  1. Your code is structured around functions
  2. Each function performs a single, specific task
  3. Functions are re-used throughout your code base
  4. Higher level functions are composed out of lower level functions
  5. Function names are a clear indication of what they do
  
- ❑ Why? Re-usable, Intuitive, Allows for clear code, Easy to extend and maintain
- ❑ Functional programming dramatically reduces code duplication

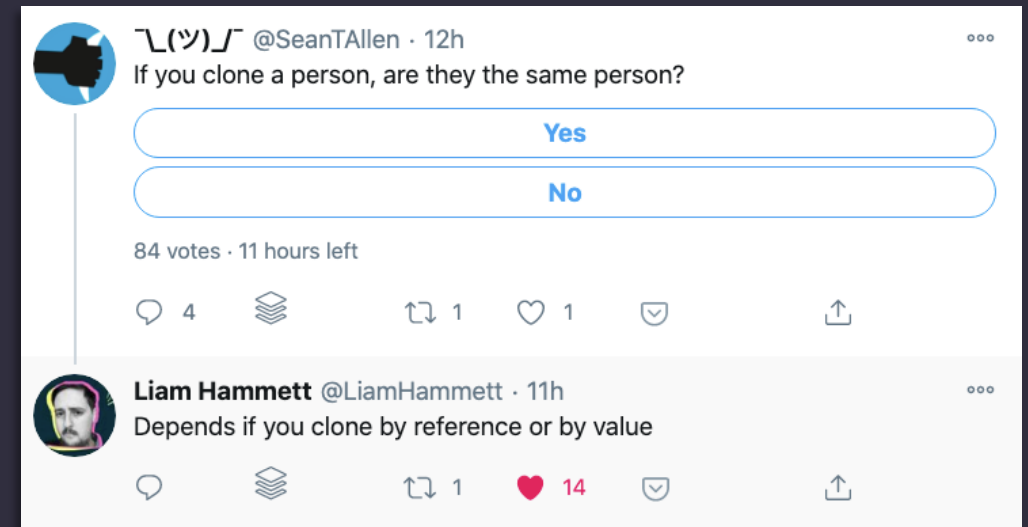
# Functions: functional programming example

```
function load_sequence(id)
  # first find a suitable download location
  download_repo = nothing
  for repo in ALL_REPOS
    if id in repo.index
      download_repo = repo
      break
    end
  end
  isnothing(download_repo) && error("No download")
  # actually download the sequence
  connect!(download_repo.connection, id)
  protein_sq = download_sq(id, download_repo)
  # Check sequence
  for aacid in "BXZJOU"
    if aacid in protein_sq
      error("Invalid sequence")
    end
  end
  return protein_sq
end
```

```
function load_sequence(id)
  repo = find_repo(id)
  protein_sq = download_sq(repo, id)
  validate(protein_sq)
  return protein_sq
end
```



Quick note: this only works when objects are passed *by reference, not by value*





# Functions: size

- ❑ Functions should be *short!* Pop quiz: how short?
- ❑ 3 to 100, with median 30!

- ❑ **Functions should have 1 level of abstraction**

- This also makes the functions cleaner, and outlines their use
- Small functions make bugs easy to find and solve

- ❑ **Duplicate code means you should be having a function instead!**

```
1  function load_sequence(id)
    seq = download_seq(id)

    # calculate the complement; exchange C-G, A-T
    complement_map = Dict(
        'A' => 'T', 'T' => 'A',
        'C' => 'G', 'G' => 'C'
    )
    complement = copy(seq)
    not_recognized = 'N'
    for i in 1:length(complement)
        n = seq[i]
        complement[i] = get(complement_map, n, not_recognized)
    end

    cleaned_seq = remove_flanking_n(complement)
    return cleaned_seq
end
```

1

2

# Functions: purpose

---

- ❑ Functions should do one thing, and one thing *only*
- ❑ Also known as the Single-Responsibility Principle
- ❑ Possible “1-things” (non-exhaustive list):
  - Calculation of quantities
  - Plotting
  - Communication with outside world (global scope)
  - Job submission on a cluster
  - Loading/reading data
  - Processing data
  - Search
  - Question (true/false)
- ❑ If you have a function that does 2 or more of the above, you gotta split it up!
- ❑ This increases the *reusability* of functions and *decreases the bug risks!*

# Functions: naming

---

- ❑ The name of a function is exceptionally important!
- ❑ Must convey the “one thing” the function does
  - Clearly, and without ambiguity!
  - Too long names, or not clear enough, are an indicator that the function doesn’t do “one thing”
- ❑ Examples: `lyapunovspectrum`, `nearby_agents`, `mutualinformation`, ...
- ❑ Important: do not “shadow” functions
  - Which means: don’t use names of existing functions, even in case of no conflict
  - E.g., ``np.sort`` and ``plt.sort`` and ``mypackage.sort`` are legitimate code due to namespace separation by the parent module
  - However this can be confusing to the reader (do the functions do the same thing?)
  - Instead, extend the base functions (e.g., via single/multiple dispatch) if they do the same thing,
  - Or define new ones whose name better reflects this different behavior

# Functions: a bad example

---

- ❑ What is the side-effect here?
- ❑ How does the function violate the “1-thing” rule?

```
function check_authentication(  
    username, password  
)  
    user = get_user(username)  
    encoded = encode(password)  
    if user.encoded_pass == encoded  
        initialize_user_session()  
        return true  
    else  
        return false  
    end  
end
```

- ❑ What does “false” mean? Is the password false, is there a network error?

# Functions: arguments

- ❑ Keep the number of arguments small: 0, 1, 2, 3. The least the better!
  - Each extra argument should be weighted carefully as it increases mental mapping when reading
- ❑ Make the non-crucial arguments either keywords, or passed in a container
  - A good example is plotting functions that use the *keyword propagation technique*
  - Another example are model parameters which are passed as a named container

```
function plot_field_cor(X, Y; kwargs...)
    z = spatial_cor(X, Y)
    color = maximum(z) > 10.0 ? "C0" : "C1"
    plot_field(z; color, kwargs...)
end
```

```
function plot_field(X;
    color = "C0", marker = "o")
    # do the actual plotting
end
```

Julia

```
def plot_field_cor(x, y, **kwargs):
    z = spatial_cor(x, y)
    color = "C0" if z.max() > 10.0 else "C1"
    plot_field(x, y, color, **kwargs)
```

```
def plot_field(x, y, color="C0", marker="o"):
    # do the actual plotting
```

Python

# Functions: main

---

- ❑ Define functions in a “main”-like setting, with the called functions being declared after the main function
- ❑ This is similar with how a scientific paper is structured
  - The model or main result is summarized
  - Everything is defined in more detail in later sections (e.g. “methods”)
  - Also like with figures: a figure is only placed after it has been referenced once
- ❑ If your language doesn’t allow this...
  - You are ***certainly*** using too old of a language

```
function load_sequence(id)
    seq = download_seq(id)
    validate(seq)
    return seq
```

```
end
```

```
function download_seq(id)
    # implementation
end
```

```
function validate(seq)
    # implementation
end
```

# Exercise: legacy code → modern clean code

---

- ❑ Get the `eratosthenis\_sieve` code (Julia and Python versions available)
  - Translate it verbatim to your language of choice if necessary
  
- ❑ Make it cleaner and more readable by:
  1. Using higher level syntax that your language provides (e.g., list comprehensions or broadcasting)
  2. Using functional programming and better naming
  3. Using functions from the standard library for e.g., counting or finding true elements
  
- ❑ What were the major changes you did?
- ❑ What comments did you have to use, if any?
- ❑ ***Max time: 20 minutes!***



# Comments

---

- ❑ “proper use of comments is to compensate for our *failure* to express ourselves in code”
  - *Clean Code: A Handbook of Agile Software Craftsmanship*
- ❑ Problems with comments:
  - Comments are hard to maintain
  - Wrong comments are *much worse* than no comments
  - Comments are expressed in human language, not code language
  - There are *better ways to document code* in human language
- ❑ *Self-explanatory & simple code is superior to complicated but commented code*

# Bad Comments

- ❑ Bad comments should be avoided at all costs! Here are some examples:

- redundant
- misleading or incorrect
- replacing proper code

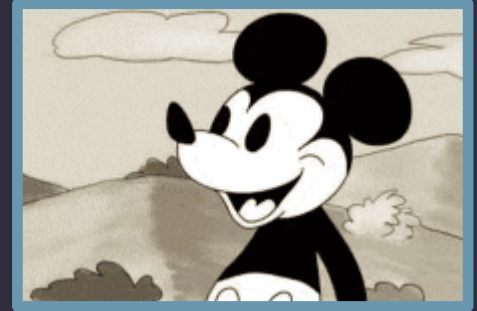
```
# validate protein sequence  
valid = validate_sequence(protein_seq)
```

```
# return true if all aminoacids are valid  
!valid && error("Invalid")
```

```
toks = split(line)  
# toks[5] contains the raw p-value,  
# toks[6] the test number  
adj_pval = calculate_adj_pval(toks[5], toks[6])
```

```
# versus:  
raw_pval = toks[5]; testnumber = toks[6]  
adj_pval = calculate_adj_pval(raw_pval, testnumber)
```

- ❑ Don't comment out code:
  - E.g. fix bug or
  - Add a slightly modified version of an algorithm below the commented one
- ❑ **Runnable but commented-out code leads to confusion!**
- ❑ Comments in full capitals are distracting & flow-interrupting



# Good Comments

- ❑ Warning of Consequences
- ❑ Explaining a Regex or other typically unfamiliar structures
- ❑ TODOs
- ❑ Justifying an operation that can only be known by knowledge of a specific paper
- ❑ Defining short-named variables in scripts
- ❑ Justifying a value choice

```
# this test requires ~1h on local machine; can't run on CI  
@test crazy = longestest(val)
```

```
# match genomic regions in the format <id>:<start>-<end>  
genomic_regex = r"([^\:]+):(\d+)-(\d+)$"
```

```
# TODO: Generalize to higher dims  
for i in 1:5  
    stuff...  
end
```

```
# Perform eqs. (19) and (20) from Datseris et al., 2019  
x = 5y^2 + 2
```

```
B = 5 # magnetic field (in Tesla)  
V = 2.5 # potential (in eV)
```

```
"good value for peak detection & alignment"  
const PEAKW = 7
```

# Best comments (?)

---

- ❑ It is nearly impossible to comment the actual behavior of code; only the intended
- ❑ **High level intention-based description of code blocks = best comments**
  - Also known as “Commenting Showing Intent”, CSI
  - Comments describe what the following block of code intends on doing
  - These comments must be language agnostic
  - Such comments are always recommended to have!
  - Example: [partially predictable chaos source code](#) from DynamicalSystems.jl
- ❑ **HOWEVER!**
  - In you follow the good practices of this workshop, you will write very few CSI! Why...?
  - In the majority of cases CSI actually become **documentation strings!** [block 5]
  - Since each function has its own docstring, and functions are small, CSI are rarely needed
  - In fact, it was surprisingly hard to find examples for this slide...
  - (Obviously, sometimes you will have to write larger functions. There CSI are very helpful!)
- ❑ To make the case: solution of the Eratosthenis sieve exercise

# Vertical Formatting: Files

Outside the file: group functionality logically into files and folders

- Start script files with a description (comment) about what the file contains

```
#=  
Styling file for the figures of the  
velocity correlation paper.  
=#
```

```
#=  
This file defines functions that simply return all timeseries of velocity  
and timing from the different sources we have. E.g. PG13, Uwe recordings,  
Drum playalongs, drum tapping, etc.  
Any extra options (like e.g. using only melody, etc.), are passed  
as keyword arguments.  
  
All functions always return (mtds, vels)  
=#
```

- Put headers into logical sections of each source file

```
#####  
# Histograms  
#####  
maxnotelevel(::Type{Note}) = 127
```

```
#####  
# Velocity peaks  
#####  
find_peaks(h) = peaks = findall(:
```

# Vertical Formatting: Code

---

Inside the file:

- ❑ Properly use blank lines, similarly with writing a paper
- ❑ Everything in the same train of thought is cohesive, without blank lines
- ❑ Blank lines separates complete thoughts (paragraphs)

# Vertical Formatting: Code

Machine-learning technology powers many aspects of modern society: from web searches to content filtering on social networks to recommendations on e-commerce websites, and it is increasingly present in consumer products such as cameras and smartphones. Machine-learning systems are used to identify objects in images, transcribe speech into text, match news items, posts or products with users' interests, and select relevant results of search. Increasingly, these applications make use of a class of techniques called deep learning.

Conventional machine-learning techniques were limited in their ability to process natural data in their raw form. For decades, constructing a pattern-recognition or machine-learning system required careful engineering and considerable domain expertise to design a feature extractor that transformed the raw data (such as the pixel values of an image) into a suitable internal representation or feature vector from which the learning subsystem, often a classifier, could detect or classify patterns in the input.

Representation learning is a set of methods that allows a machine to be fed with raw data and to automatically discover the representations needed for detection or classification. Deep-learning methods are representation-learning methods with multiple levels of representation, obtained by composing simple but non-linear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level. With the composition of enough such transformations, very complex functions can be learned. For classification tasks, higher layers of representation amplify aspects of the input that are important for discrimination and suppress irrelevant variations. An image, for example, comes in the form of an array of pixel values, and the learned features in the first layer of representation typically represent the presence or absence of edges at particular orientations and locations in the image. The second layer typically detects motifs by spotting particular arrangements of edges, regardless of small variations in the edge positions. The third layer may assemble motifs into larger combinations that correspond to parts of familiar objects, and subsequent layers would detect objects as combinations of these parts. The key aspect of deep learning is that these layers of features are not designed by human engineers: they are learned from data using a general-purpose learning procedure.

```
3016 # we are done
3017 if isinstance(self.grouper, Grouping):
3018     self.grouper = self.grouper.grouper
3019
3020 # no level passed
3021 elif not isinstance(self.grouper,
3022                     (Series, Index, ExtensionArray, np.ndarray)):
3023     if getattr(self.grouper, 'ndim', 1) != 1:
3024         t = self.name or str(type(self.grouper))
3025         raise ValueError("Grouper for '%s' not 1-dimensional" % t)
3026     self.grouper = self.index.map(self.grouper)
3027     if not (hasattr(self.grouper, "__len__") and
3028            len(self.grouper) == len(self.index)):
3029         errmsg = ('Grouper result violates len(labels) == '
3030                  'len(data)\nresult: %s' %
3031                  pprint_thing(self.grouper))
3032         self.grouper = None # Try for sanity
3033         raise AssertionError(errmsg)
3034
3035 # if we have a date/time-like grouper, make sure that we have
3036 # Timestamps like
3037 if getattr(self.grouper, 'dtype', None) is not None:
3038     if is_datetime64_dtype(self.grouper):
3039         from pandas import to_datetime
3040         self.grouper = to_datetime(self.grouper)
3041     elif is_timedelta64_dtype(self.grouper):
3042         from pandas import to_timedelta
3043         self.grouper = to_timedelta(self.grouper)
3044
```



# Horizontal Formatting

- ❑ Lines should always have a maximum character length
  - 80, 88 or 92 are common guidelines, I use 92, but check your style guide
- ❑ Each new code block (for loops, functions, ...) introduces a new level of indentation
  - Use 4 spaces instead of 2 and *never use TAB*
- ❑ When writing floating point literals, they should always include a leading and/or trailing zero if necessary

# Yes:	# No:
0.1	.1
2.0	2.
3.0f0	3.f0

```
function flaggedcells(gameboard)
→ flagged = []
  for cell in gameboard
    → if cell.status == FLAGGED
      → push!(flagged, cell)
    end
  end
  return flagged
end
```

# Spacebar: it's your friend!

- ❑ White spaces (spacebar) are typically defined by the style guide.
- ❑ Strongly recommend to follow a style guide with spacebars **everywhere!**
  - All binary operators have enclosing white spaces, except of: `*`, `^`, `/` (because of math syntax)
  - Spaces after any commas and semicolons `(, ;)`
  - Assignment `(=)` enclosed by spaces always, keyword assignment depends on your style guide

```
b*b-4*a*c      # no
b * b - 4 * a * c # no
b*b - 4*a*c     # yes!
```

```
a=b&&(x.<y)/(z ^ 2)      # no
a = b && (x .< y)/(z^2)  # yes!
```

```
f(x,y;z=3)      # no!
f(x, y; z = 3)  # yes!
f(x, y; z=3)    # depends on style, and typically preferred in Python!
```

# Don't reject your friends!

```
uhtraj,=ivp((ta,ua)->fxp(ua,yphet[3]),[uini[1],yphet[1]],[0,yphet[2]],N;
    output="trajectory")
ua0,conva=newton(x->fxp([x[1];0],phet0),[minimum(uhtraj>>1)])
ub0,convb=newton(x->fxp([x[1];0],phet0),[maximum(uhtraj>>1)])
evec(u,p,num)=
begin
    J=jacobian(ua->fxp(ua,p),[u[1],0])
    ev=eigen(J)
    iev=sortperm(real.(ev.values))
    vec=ev.vectors[:,num]
    vec=vec*sign(vec[2])/norm(vec)
    return vec
end
va0,vb0=evec(ua0,phet0,1),evec(ub0,phet0,2)
offset=1e-3
tguess=50
Nhet=300;
xmid=(ua0[1]+ub0[1])*0.5
M0=(t0,u0,p,sgn)->ivp((ta,ua)->fxp(ua,p),u0,[0,t0],Nhet,
    output="trajectory",stop=u->(u[1]-xmid)*sgn>0)
utha,tha=M0(-tguess,[ua0[1],0]+offset*va0,phet0,1)
```

this codeblock is *incredibly difficult to parse at a glance*. For two reasons: bad variable naming, but most importantly, the whole codeblock has only a single space bar in 753 characters (and that's the mandatory after return).....

It's like the words are mushed into each other; imagine trying to read a book without spaces. Doesn't make sense right? Well the same goes for code...

# Consistency above all

---

- ❑ So far: much advice on how to write clean code. Important to follow!
- ❑ Equally important: consistency across a file, then across a repo, then across an organization!
- ❑ Strongly recommended: follow a Style Guide and (if possible) incorporate an automated formatter for your code!
- ❑ Even more strongly recommended: agree with your co-developers on the style!
  
- ❑ A Julia recommendation:
  - <https://github.com/domluna/JuliaFormatter.jl> with <https://github.com/SciML/SciMLStyle>
- ❑ A Python recommendation:
  - <https://github.com/psf/black>

# Your code → clear code

---

1. Clean up your ~~mess~~ code!
  - a. Establish good naming conventions throughout your code's variables
  - b. Remove all bad comments and only leave good and necessary ones
2. Functional programming
  - a. Identify large functions in your code base
  - b. Separate functions into smaller chunks by making functions call other functions
  - c. Ensure all functions have one purpose
3. File organization
  - a. Separate your code logically into folders and files
  - b. Write a description for each file (comment at the top summarizing the file's contents)
  - c. Provide a logical organization of the file's structure by grouping functions with similar/connected functionality in sections, each with its own header
4. At the end, add a new tag at this stage called “clear” (remember that all of these changes should be done via a branch + merge!)