# #3-3: Algorithmic Thinking and Style

CS SCHOLARS – PROGRAMMING

# Learning Goals

Identify whether a problem can be solved by **following an algorithm**, **applying a pattern**, or **problem solving**

Apply general **style principles** to write **clear** and **robust** code

# Solving Familiar Problems

# Designing Algorithms

Finding the right Python syntax to write a program isn't really the hardest part of programming. Designing an algorithm to solve a problem is where the process really gets tricky.

Luckily, designing new algorithms doesn't always have to be difficult. There are many situations where you don't need to make a new algorithm at all; you can either use a **pre-existing algorithm**, or you can use an **algorithmic pattern** you've used before.

# Pre-Existing Algorithms

Algorithms can be represented in many forms other than code!

Sometimes an algorithm might be provided to you in the text of the problem statement itself.

For example, consider the prompt to the right. We can use this prompt to find the **input** (a number), the **output** (a Boolean, whether it's prime), and a bit of the **algorithm** (check that only 1 and the number itself are factors).

Algorithms can also be provided as flow charts, or formulas.

Write a function to determine whether a given number is prime.

A number is prime if it is only divisible by two numbers – itself and 1.

No other number between 1 and the prime should be a factor.

Note that primes must be larger than 1.

# Pseudocode

If we wanted to break this into abstracted algorithmic steps (also called **pseudocode**), it might look like this:

**Input:** int (the number, x)

**Output:** bool (whether it's prime)

1) Verify that x is bigger than 1

2) Check all the numbers between 1 and x

   a) Verify that the number does not evenly divide x

3) Output whether x passed all verifications

It's often easier to code an algorithm if you write out the pseudocode first!

# Algorithmic Patterns

There will also be many cases where an algorithm is not directly available for a program you need to implement, but you still don't need to invent a brand-new algorithm on your own. These programs will often follow common **algorithmic patterns**.

If you can recognize when a problem is similar to a pattern you've seen before, you can make the problem-solving process much more straightforward.

# Example Algorithmic Pattern

For example, consider the prompt on the right.

The prompt isn't giving us the exact way to solve the problem. But it feels similar to a problem we've seen before- it's checking for numbers that divide the given number, just like `isPrime`!

We can use the structure of `isPrime` (looping over possible factors and checking something for each) as a starting place.

Write a function that checks whether a number is **powerful**.

A positive integer x is powerful if, for every prime y that divides x, $y^2$ also divides x.

# Solving New Problems

# Inventing New Algorithms

There will be times when you need to write an algorithm that is unlike any problem you've worked with before. In this situation, you can't rely on a pre-built algorithm or adapt an algorithmic pattern; you need to build a new algorithm on your own.

This is one of the hardest parts of the programming process, because you can't follow instructions to get the right answer; every problem is different. This just takes practice to learn.

However, there are a few strategies you can use that might make the problem-solving process easier.

# Strategy 1: Human Computer

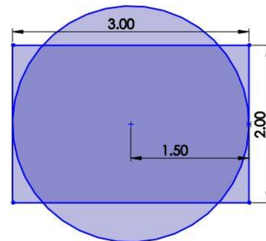
First, you can use the **human computer** strategy to look for natural approaches towards solving the problem.

Try to solve an example input to the problem yourself, as a human being. You can then extrapolate from your own approach to come up with an algorithm.

The human computer approach works best when you make your own approach as systematic and detailed as possible. Pay attention to every step you take, and make sure not to skip steps. Think about how the computer would see the problem- would the computer see it differently from you?

# Human Computer Example

**Example:** Write the function `rectangularPegRoundHole(r, w, h)`, which returns `True` if a rectangular peg with width `w` and height `h` can pass through a round hole with radius `r`, and `False` otherwise.

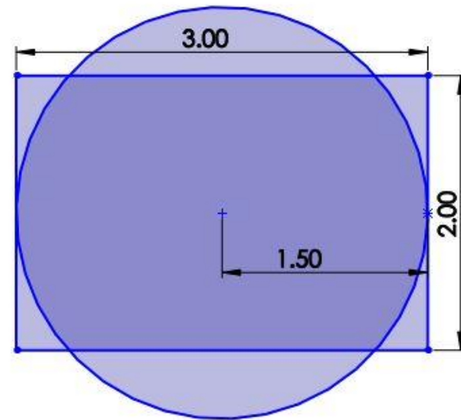



**You Do:** Imagine trying to fit a rectangular object into a round hole. How can you tell if the peg will be able to fit or not?

# Human Computer Example

The longest part of the rectangle is its **diagonal**. If the diagonal fits, the rest will too; if the diagonal doesn't fit, then the whole thing doesn't work.

Now you just need to calculate the length of that diagonal in code and compare it to the diameter.



# Human Computer Code

```
def rectangularPegRoundHole(r, w, h):
    # calculate diagonal
    diagonal = (w**2 + h**2)**0.5
    # calculate diameter
    diameter = r * 2
    # compare
    return diagonal <= diameter
```

# Strategy 2: Test-Driven Design

Another strategy that can help make problem solving easier is **test-driven design**. This is an approach where you start by generating test cases instead of by jumping right into the problem.

Test-driven design can be useful because it helps you think through all the requirements of the code, which can help you notice patterns and edge cases in advance. This is better than realizing you've made a logical error only after you've written all the code.

# Test-Driven Design Example

**Example:** Write the function `nearestBusStop(street)` that takes a non-negative integer street number and returns the nearest bus stop to the given street. Buses stop on every 8th street, including street 0, and ties go to the lower street.

**You Do:** what are some test cases we could use for this function that would inform us about how it works?

# Test-Driven Design Example

**Normal case:** the nearest bus stop to 6th street would be 8th street

**Edge case:** where is there a change in results, maybe from 8th street to 16th street? 12th street goes to 8th street, but 13th street goes to 16th street

**Special case:** do we need to deal with negative or float street numbers? No- the prompt says non-negative integer.

The test functions show that this is like a **step** function. We can either use conditionals or the mod operator to make this work.

# Test-Driven Design Code

```
def nearestBusStop(street):
    # get distance from prev street
    belowDistance = street % 8
    if belowDistance <= 4: # edge case specifies this
        return street - belowDistance # lower street
    else:
        return street + (8 - belowDistance) # upper
    # OR
    offset = street + 3
    return offset - (offset % 8)
```

# Strategy 3: Simplify and Solve

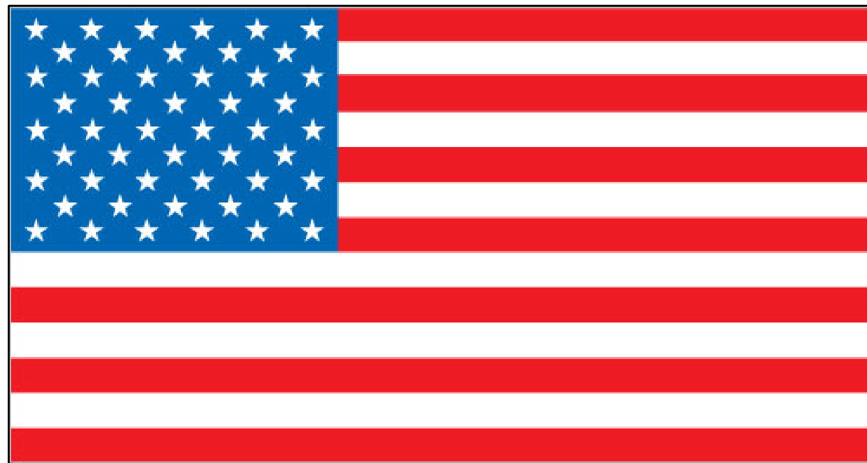
The third strategy is called **simplify and solve**. The main idea is that it's sometimes easier to solve a problem if you make that problem simpler first.

Solve the smaller problem, then add back in the more complex details once the core problem is done.

# Simplify and Solve Example

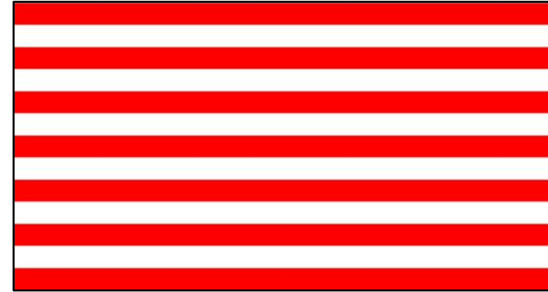
**Example:** we want to draw the flag of the United States using tkinter graphics.

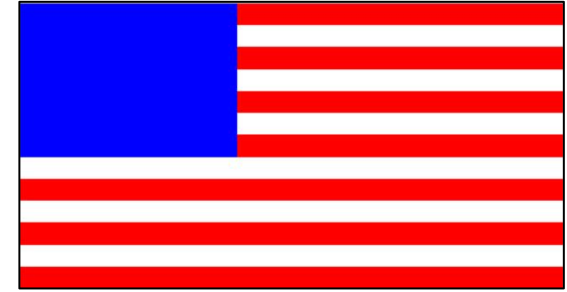
**You Do:** how can you break the US flag down into simpler parts?



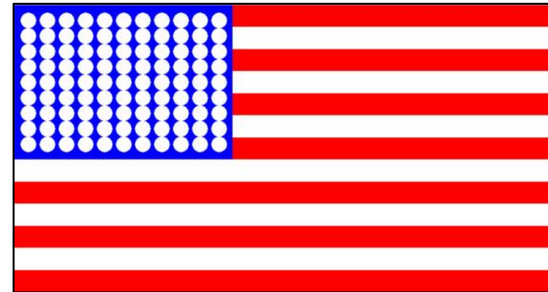
# Simplify and Solve Example

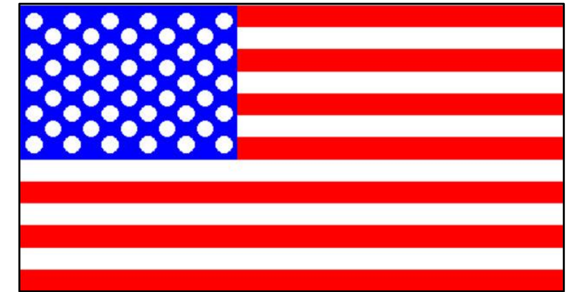
Start with just the stripes of the flag with a blue rectangle in the corner. Match proportions as you go!



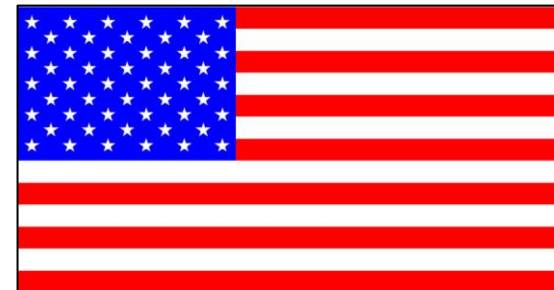


Then arrange the stars by drawing circles instead. Start with a normal grid, then alternate stars.





Finally, figure out how to draw a star instead of a circle. You can use a helper method here!



# Simplify and Solve Code – Step 1

We'll need lists to draw stars properly. But we can do the rest now!

```
def drawFlag(canvas, width, height):
    numStripes = 13
    stripeHeight = height / numStripes
    for stripe in range(numStripes):
        top = stripe * stripeHeight
        if stripe % 2 == 0:
            color = "red"
        else:
            color = "white"
        canvas.create_rectangle(0, top, width, top + stripeHeight,
                                fill=color, width=0)

    squareHeight = stripeHeight * 7
    squareWidth = width * 0.4
    canvas.create_rectangle(0, 0, squareWidth, squareHeight,
                            fill="blue", width=0)
```

# Simplify and Solve Code – Step 2

```
...
starRows = 9
starCols = 11
starYMargin = squareHeight / 20
starSize = (squareHeight - 2 * starYMargin) / starRows
innerXMargin = squareWidth / 60
outerXMargin = (squareWidth - starCols * starSize - innerXMargin * 10) / 2
for row in range(starRows):
    top = starYMargin + row * starSize
    for col in range(starCols):
        if (row % 2 == 0 and col % 2 == 0) or \
           (row % 2 == 1 and col % 2 == 1):
            left = outerXMargin + col * (starSize + innerXMargin)
            canvas.create_oval(left, top, left + starSize, top + starSize,
                               fill="white", width=0)
```

# Style

# Real-World Coding

When you're working on a homework assignment, you probably mainly care about getting the code to work.

But this isn't how programming works in real life. If you write a piece of code that accomplishes a task, it's highly likely that you or someone else will want to use that code again at some point in the future.

It's even possible that you'll want to change the code slightly when the goals of the task change.

# Purpose of Style

Whenever you write code that anyone (including yourself) will look at again in the future, you should write that code with good style.

Style is all the decisions you make as you write code about how to organize and implement an algorithm.

It's very much like a writing style or a drawing style; everyone will approach how they organize their code a little differently.

# Different Styles

```python
# Input: int
# Output: bool
def is_prime(num):
    if num < 2:
        return False
    for factor in range(2, num):
        if num % factor == 0:
            return False
    return True
```

```python
def isPrime(x):
  """
  takes an integer and returns
  whether or not it's prime
  """
  if(x<=1):
    return(False)
  # check each possible factor
  for i in range(2,x):
    if((x%i)==0):
      return(False)
  return(True)
```

# Style Principles

There are lots of recommendations for how to write code with good style. We'll group them into two major categories:

- **Clarify** – principles that make your code easier to read
- **Robustness** – principles that make code easier to modify

# Style – Clarity

# Style Principles for Clarity

You spend as much time reading code as you do writing code, if not more! Writing code that is **clear** and easy to read is therefore extremely important.

We'll look at four general principles for writing clear code:

1. Use consistent formatting
2. Use good naming conventions
3. Don't include unnecessary code
4. Remember to document

# Consistent Formatting

In general, try to be **consistent** with how you format whitespace in code.

Python will let you get away with somewhat uneven indentation in different parts of a program, but the result is harder to read. Be consistent about whether you use spaces or tabs, and always use the same number of spaces or tabs when indenting code.

**BAD**

```
def isPrime(x):
    if x < 2:
      return False
    for factor in range(2, x):
     if x % factor == 0:
             return True
    return False
```

**GOOD**

```
def isPrime(x):
    if x < 2:
        return False
    for factor in range(2, x):
        if x % factor == 0:
            return True
    return False
```

# Consistent Formatting

Code is also easier to read when the whitespace used between tokens is consistent.

You can choose to use no unnecessary whitespace, or add a space between every pair of tokens, or even choose some operators that will have whitespace added and some that won't.

| **BAD** | **GOOD** |
|---|---|
| `x =(3+   2 )    / 5` | `x = (3 + 2) / 5` |

Also- don't let your lines of code get too long. A general guideline is to pick a number of characters - let's say 80 - and make sure every line of code you write is no longer than that length. Pyzo lets you place an indicator in the editor at that location.

# Good Naming Conventions

It's important to give your variables descriptive names that describe the data held by the variable. Having descriptive, meaningful names will make understanding code much easier.

**BAD**

```
def isPrime(a):
    if a < 2:
        return False
    for b in range(2, a):
        if a % b == 0:
            return True
    return False
```

**GOOD**

```
def isPrime(num):
    if num < 2:
        return False
    for factor in range(2, num):
        if num % factor == 0:
            return True
    return False
```

There are a few cases where seemingly-meaningless variable names have gained meaning over time, usually when they are shorthand for a longer word. For example, we often use x, y, and n for numbers. These are okay to use when there's no greater meaning behind the variable.

# Avoid Unnecessary Code

Unnecessary code is code that will never actually be run by Python, or code that Python runs but never uses in a meaningful way (like a variable that is defined, then never used). We also refer to this as dead code.

**BAD**

```
def isPrime(num):
    if num < 2:
        return False
    end = num
    for factor in range(2, num):
        if num % factor == 0:
            return True
        else:
            pass
    return False
    return
```

**GOOD**

```
def isPrime(num):
    if num < 2:
        return False
    for factor in range(2, num):
        if num % factor == 0:
            return True
    return False
```

Unnecessary code won't actually harm your program, but it does make the program more complicated to understand.

# Document Your Code

Finally, make sure to document your code using comments! We haven't talked much about *when* to write comments. In general, comments are most useful when they explain something that is not immediately obvious from the code itself.

Consider this code snippet, from isPrime. This is a good comment because it clarifies something that might not be immediately obvious- we intentionally skipped num because it's okay for a prime number to divide itself.

**GOOD**

```
def isPrime(num):
    if num < 2:
        return False
    # do not iterate over 1 or num because prime
    # numbers are divisible by themselves and 1
    for factor in range(2, num):
        if num % factor == 0:
            return True
    return False
```

# Activity: Find Style Errors

**You Do:** What are some **clarity** style errors in this piece of code?

```
def sumToN(n):
    tmp = 0
    for abc in range(n):
        tmp += abc
        abc=abc+1
    return tmp
```

# Style – Robustness

# Style Principles for Robustness

There are also several principles that will help you write **robust** code. This will make your code easier to change and update over time, and decrease the chance of bugs occurring.

We'll look at four general principles for writing robust code:

1. Avoid repetitive code
2. Avoid magic numbers
3. Join up related conditionals
4. Test all functions

# Avoid Repetitive Code

First, try not to write **repetitive** code. This is code where similar logic is repeated over many lines instead of being condensed into a single structure.

When you find yourself repeating code- and **especially** when you find yourself copying and pasting code – look for the pattern and move it into a loop or a generalized action. Helper functions can be useful here too.

**BAD**

```
def coordToRow(x):
    if x < 50:
        return 0
    elif x < 100:
        return 1
    elif x < 150:
        return 2
    elif x < 200:
        return 3
```

**GOOD**

```
def coordToRow(x):
    for row in range(4):
        if x < row * 50:
            return row
```

**ALSO GOOD**

```
def coordToRow(x):
    return x // 50
```

# Avoid Magic Numbers

Second, avoid using **magic numbers**. These are numbers used somewhere in an algorithm for no clear reason, without being stored in a variable first.

Magic numbers are mainly a problem when it comes to updating code. Consider what would be required to change the size of the grid cells in these two implementations.

**BAD**

```
def drawGrid(canvas, canvasSize):
    for row in range(4):
        top = row * 50
        bottom = top + 50
        for col in range(4):
            left = col * 50
            right = left + 50
            canvas.create_rectangle(left, top,
                                    right, bottom)
```

**GOOD**

```
def drawGrid(canvas, canvasSize):
    rows = 4
    cellSize = canvasSize / rows
    for row in range(rows):
        top = row * cellSize
        bottom = top + cellSize
        for col in range(rows):
            left = col * cellSize
            right = left + cellSize
            canvas.create_rectangle(left, top,
                                    right, bottom)
```

# Non-Magic Numbers

Not every number is a magic number. For example, to get the ones digit of a number you have to mod by 10. In this case, it's pretty clear why 10 is used, and you're not likely to change it to anything else in the future. So you don't need to store 10 in a variable.

```
def getOnesDigit(num):
    return num % 10
```

0, 1, 2, and 10 are often (though not always) safe to use directly in code.

# Join Up Conditionals

Third, make sure to **join up conditionals** as appropriate. If you have multiple conditional checks that are happening in a row and only one of them should be visited, those checks should form one `if`-`elif`-`else` block, not several independent `if`s.

**BAD**

```
def getSize(length):
    size = ""
    if length <= 38:
        size = "small"
    if 38 < length <= 40:
        size = "medium"
    if 40 < length:
        size = "large"
    return size
```

**GOOD**

```
def getSize(length):
    size = ""
    if length <= 38:
        size = "small"
    elif length <= 40:
        size = "medium"
    else:
        size = "large"
    return size
```

The main problem with not joining up related conditionals is that you might accidentally enter more than one conditional branch if you aren't careful with the tests. It's just safer to combine them all together.

# Test Your Functions

Finally, make sure to **write test functions** for each function you implement! Yes, writing test cases takes time and can be tedious, but it will help you out a lot in the long run.

Test functions are primarily useful at two points in time. The first is naturally when you first write a function. The test function ensures that it's working the way you want it to.

But test functions are also useful later on, if you need to modify a function. Having an active test suite makes it easy to check whether a new modification breaks any of the previous requirements of the program.

# Activity: Find Style Errors

**You Do:** What are some **robustness** style errors in this piece of code?

```
def getSize(length):
    size = "small"
    if 38 < length and length <= 40:
        size = "medium"
    if 40 < length:
        size = "large"
    return size
```

# Learning Goals

Identify whether a problem can be solved by **following an algorithm**, **applying a pattern**, or **problem solving**

Apply general **style principles** to write **clear** and **robust** code