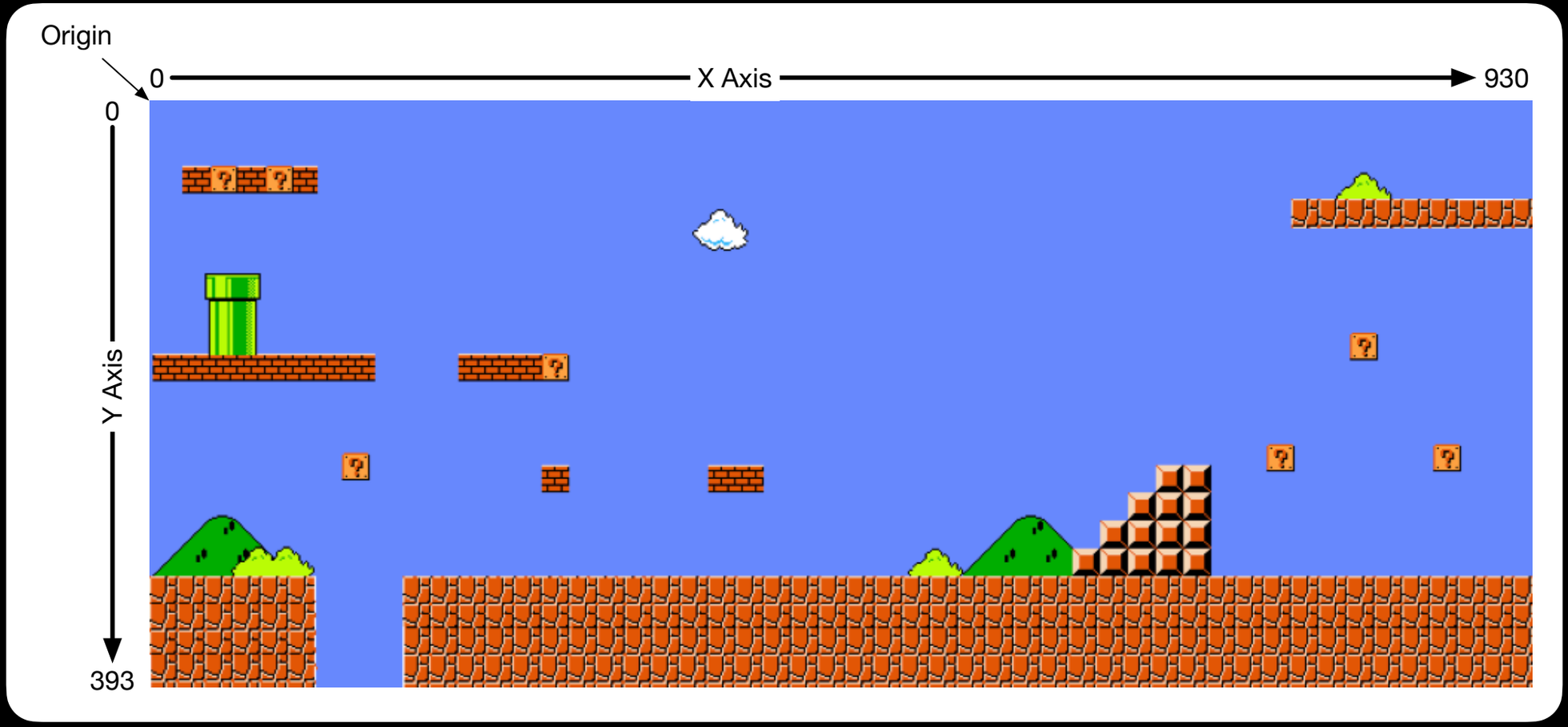
Algorithms

(for Game Design)

Session 2: More Pygame, Sprites and Randomness!

last session

A Digital Image



The Surface

- In Pygame, a **surface** is a place to draw
  - Think of it as a canvas on which an artist places paint
  - Or, as a bank of memory where all those pixels are stored
- Special surface: the **display surface** -- will be visible to the user in the game window

Returned from `set_mode` method as:

```
display_surface = pygame.display.set_mode((1024, 768))
```

More Draw module

```
pygame.draw.lines(surface, color, closed, points, width=1)
```

- `lines` will draw a series of lines, consecutively connecting (x,y) tuples in the `points` list
- If `closed` is `True`, will also connect the last point back to the first

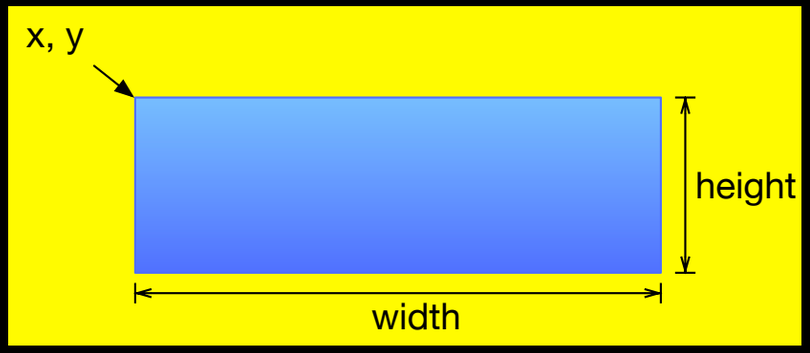
```
pygame.draw.polygon(surface, color, points, width=1)
```

- polygon acts similarly, connecting points from a list, but then it fills in the enclosed space

```
pygame.draw.circle(surface, color, center, radius)
```

- Drawing a circle is pretty obvious

Rect



- Pygame very, very commonly uses an object called a `Rect` to describe a box
  - The `arc` function used it as a bounding box
- Rects are defined by four values
  - `Rect(x, y, width, height)`
  - or less commonly, by two pairs of tuples
  - `Rect((x, y), (width, height))`

Pygame Blit

- The `Surface` object has a `blit` method for pasting another image (or portion) into the image

```
surface.blit(source, dest, area=None, special_flags=0)
```

  - The `source` parameter is another surface
  - `dest` is a location: (x, y) tuple or Rect with the top-left location you want
  - `area` can be a Rect that will specify a smaller portion of the source surface to be pasted
    - If `None`, then the entire source surface is pasted

fonts and text renders

Text

- Text is sometimes useful in a game
  - Narrations or lengthy instructions
  - Labels for objects, persons, places
  - Very stylized text is often just an image
    - i.e. That scroll with the parchment backing and roughed edges

Font Module

- Pygame renders text to a surface with the Font module
- `pygame.font` is initialized with `pygame.init()`
  - Can separately initialize with `pygame.font.init()`
- Font module has methods to locate and load a font as a `Font` object
  - Uses TrueType or FreeType fonts
- Most of the work is done by the `Font` object

```
# Example of available fonts with pygame.font
>>> pygame.font.init()
>>> available = pygame.font.get_fonts()
>>> print(f'There are {len(available)} fonts available')
>>> available[:10]
['kokonor', 'stixintupbol', 'helvetica',
'opensanssemibold', 'tsukushiamarugothic',
'microsoftsansserif', 'sfcompacttextitalic',
'stixsizthreesymbol', 'papyrus', 'bradleyhandbold']
```

Takes a while (~3 sec)

383 on my machine

Notice: unsorted, lower-case, no spaces, [types appended]

```
# Load a font, assuming I know the pygame name
font_size = 18
font_name = 'bradleyhandbold'
f = pygame.font.SysFont(font_name, font_size)
    # Extra Boolean parameters for bold and italic

# Can also pass in a list of font strings
font_name = ['bradleyhandbold','helvetica']
# Can also just get the default pygame font
font_name = None
```

New in 2.0.1: pass in an iterable

```
def setup_fonts(font_size, bold=False, italic=False):
    ''' Load a font, given a list of preferences

        The preference list is a list of strings (should probably be a parameter),
        provided in a form from the FontBook list.
        Any available font that starts with the same letters (lowercased, spaces removed)
        as a font in the font_preferences list will be loaded.
        If no font can be found from the preferences list, the pygame default will be returned.

        returns -- A Font object
    '''
    font_preferences = ['Bangers', 'Iosevka Regular', 'Comic Sans', 'Courier New']
    available = pygame.font.get_fonts()
    prefs = [x.lower().replace(' ', '') for x in font_preferences]
    for pref in prefs:
        a = [x
             for x in available
             if x.startswith(pref)
            ]
        if a:
            fonts = ','.join(a) #SysFont expects a string with font names in it
            return pygame.font.SysFont(a, font_size, bold, italic)
    return pygame.font.SysFont(None, font_size, bold, italic)
```

List specifies preferred order

list comprehension: create a list of font names found in available, but only those that start with this pref

If no preferred font found, return default

Font Object from File

- An alternate way to get a font object is to use the constructor

```
pygame.font.Font(filename, font_size)
pygame.font.Font(file_object, font_size)
```

- The constructor takes a filename string or a python file object
- Probably not worth figuring out the path to system fonts in order to use this way
- Useful if you are distributing a font with the game

The Font Object

- Great! You've got a font object. What do you do with it?

```
render(text, antialias, color, background=None)
```

- Returns a new `Surface`, with your text drawn on it
- Can only be a single line of text
- If antialias is `True`, the text will be smoothed at the edges
- You can specify `color` of text and the `background`
- Transparent background unless one is specified

a_font.size(text)

- Returns a tuple of `(width, height)` specifying the dimensions required to render the text
  - Very specific to the characters in the text string. Most fonts kern, which adjusts the width of each character for specific letter pairs
  - i.e. "ae" generally takes less space than "a" + "e"
- Note: you could just render the text and then use `get_size()` on the resulting surface -- but, this is faster
  - If you are trying to word wrap a paragraph, which would involve lots of size tests, use size instead of render for those tests

```
a_font.get_height()
```

- Returns the height of the font (the average size for each glyph in the font)

```
a_font.get_linesize()
```

- Returns the height for a line of text with the font.
    - This is the spacing between multiple lines of text

```
a_font.metrics(text)
```

- Returns a list with one tuple per character in the text string
    - Each tuple has information on exact sizes and placement of that character

```
see also: get_ascent(), get_descent()
```

```
# Replace the logo bouncer with a text bouncer
    pygame.init()

    size = width, height = 1024, 768
    speed = [3,2]
    black = (0, 0, 0)
    white = (255,255,255)

    screen = pygame.display.set_mode(size)
    pygame.display.set_caption('Text Bouncer')
    text = input('What would you like to say? ')

    font = setup_fonts(48)
    text_surface = font.render(text, True, white, black)
    text_width, text_height = text_surface.get_size()
    text_x = text_y = 0

    ....

    screen.fill(black)
    screen.blit(text_surface, (text_x, text_y))
    pygame.display.flip()
```

You specify the text

Our font loader

render

size used for bounce calculations

blit from the rendered surface to the display

sprites

OO

- You can imagine that the elements of your game (monsters, aliens, lasers, ...) can be drawn with these techniques
  - Each may be an object and you may use classes to organize a hierarchy of these elements
- Pygame can help with the organization

Sprite

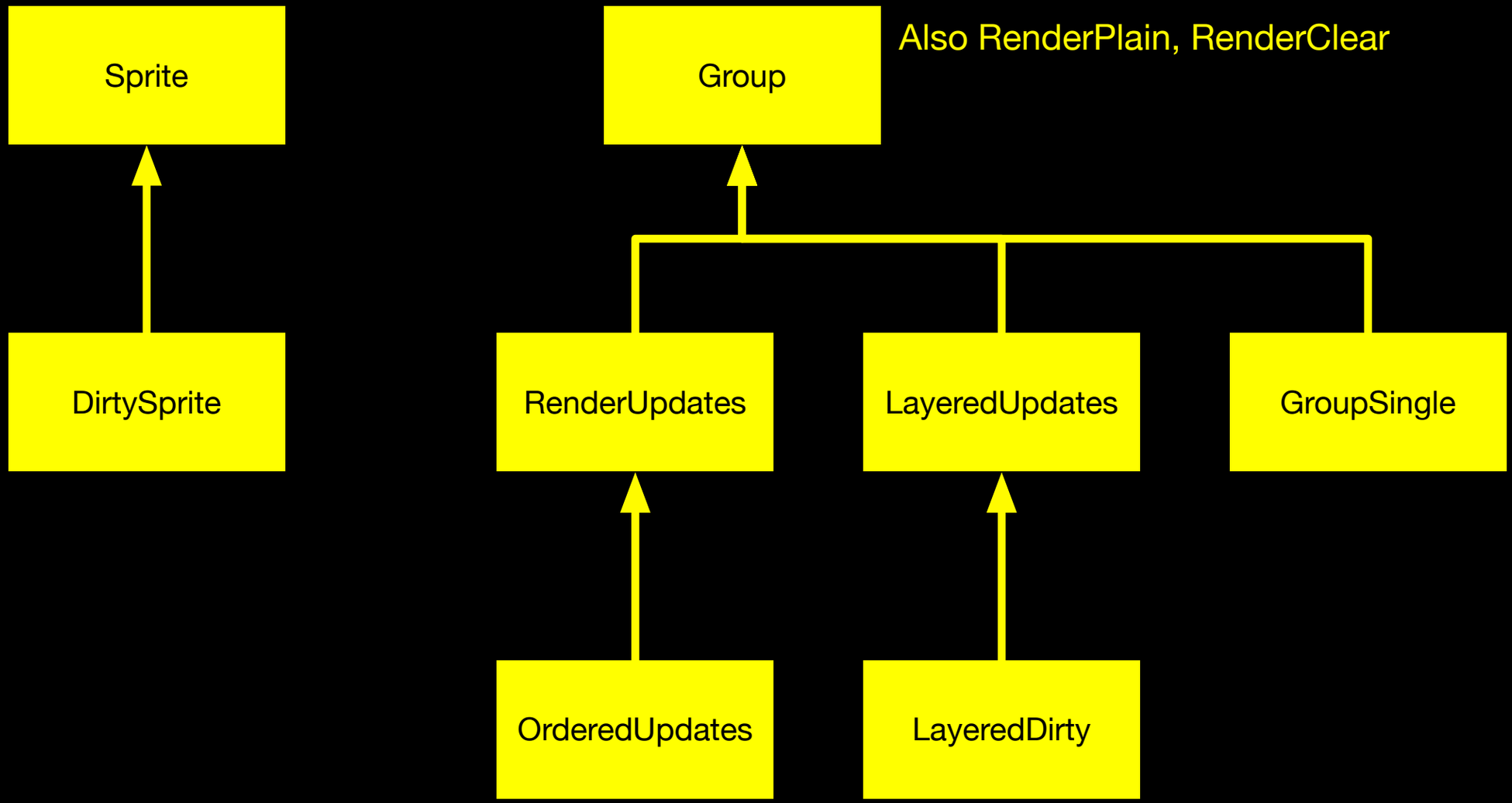
- Pygame's `Sprite` class is intended to assist to make objects of the on-screen elements
- A fairly simple base class
  - methods for updating and drawing the sprite
- `Group` classes (there are several) are useful to manage your sprites
  - Create one group for all the aliens and another for all the laser beams
  - when `update( )` is called on the group, it iterates through all the `Sprite`s in the group
  - when `draw()` is called on the group, all the `Sprite`s will be blitted, based on their `image` and `rect` attributes

```
class Mario(pygame.sprite.Sprite):

    # Constructor.  Pass in the location of Mario
    def __init__(self, x, y):
        super( ).__init__(self)
        self.x = x
        self.y = y
        self.image = pygame.image.load('mario_sprites.png')
        self.rect = self.image.get_rect()

    def update(left, right, jump):
        if left:
            self.x += ....
```

pygame.sprite Class Hierarchy



Sprites First

- `pygame.sprite.Sprite` should always be subclassed
  - Your subclass should have `image` and `rect` attributes
- Makes sense: A sprite is an on-screen object with an image and a position
- Override the `update()` and `draw()` methods for your chosen way of updating / drawing

```
# sprite1.py
class Mario(pygame.sprite.Sprite):

    def __init__(self, pos):
        pygame.sprite.Sprite.__init__(self)

        self.image = pygame.Surface((29,47))
        image_surf = pygame.image.load('mario_sprites.png').convert()
        self.image.blit(image_surf, (0,0), (4, 13, 29, 47))
        self.rect = self.image.get_rect(topleft=pos)

    def update(self, click):
        if click:
            self.rect.center = click

    def draw(self, surface):
        surface.blit(self.image, self.rect)
```

subclass of Sprite

Its own small surface for the image

Manages its own drawing process

```
# What does the game loop look like?
....
mario = Mario((100, 200))

while True:
    quit, click = check_events()
    if quit:
        break

    mario.update(click)

    surface.blit(background, (0,0))
    mario.draw(surface)
    pygame.display.flip()
```

Make a Mario at the initial position

click is either None or (x,y) of the mouse click

move mario to a new location (if clicked)

tell mario to draw himself on the window

Sprite

- Sprites are fairly simple:
  - Subclass them, but your code manages updates (and, perhaps) drawing
- The rest of the behavior has to do with how they interact with Groups (coming next)

```
Sprite.add(*groups)

Sprite.remove(*groups)

Sprite.kill()  # Remove from all groups

Sprite.alive()  # Returns True if belongs to 1+ groups

Sprite.groups()  # List of groups that contain this sprite
```

```
# sprite2.py: Draw Marios with Groups
class Mario(pygame.sprite.Sprite):

    def __init__(self, pos):
        pygame.sprite.Sprite.__init__(self)

        self.image = pygame.Surface((29,47))
        image_surf = pygame.image.load('mario_sprites.png').convert()
        self.image.blit(image_surf, (0,0), (4, 13, 29, 47))
        self.image.set_colorkey((255,255,255))
        self.rect = self.image.get_rect(center=pos)

    def update(self, size_y):
        self.rect.move_ip(0,1)
        if self.rect.top > size_y:
            self.kill()
```

Mario class is very similar to V1

Except, no draw() method

And, I made a fancier update

Where did size_y come from?

```
#... but the Game Loop has differences
mario_group = pygame.sprite.Group()

while True:
    quit, click = check_events()
    if quit:
        break
    if click:
        mario = Mario(click)
        mario_group.add(mario)

    mario_group.update(window_size_y)

    surface.blit(background, (0,0))
    mario_group.draw(surface)
    pygame.display.flip()
```

A group to hold many marios

If mouse is clicked, make a mario there

and add it to the group

Update the group (not the sprite)

parameter passed through

Draw the group (not the sprite)

Efficient Frame w/Groups

- The `Group` can also be used to more efficiently draw the frame
- Before: Every game loop, start with nothing, blit the background, blit each sprite
- Now: Every game loop, just erase where the sprites were, then update them, then redraw them
- `Group.clear(surface_dest, background)` will draw from the background into the destination at every place a sprite was last drawn
- Obviously, if your background animates, you can't do this

```
# sprite3.py:
surface = pygame.display.set_mode([window_size_x,window_size_y])
pygame.display.set_caption('Mario as a Sprite')
background = pygame.image.load('mario_background.png').convert()
surface.blit(background, (0,0))

mario_group = pygame.sprite.Group()

while True:
    quit, click = check_events()
    if quit:
        break
    if click:
        mario = Mario(click)
        mario_group.add(mario)

    mario_group.update(window_size_y)
    mario_group.clear(surface, background)
    mario_group.draw(surface)
    pygame.display.flip()
```

blit the background once

update all the sprites

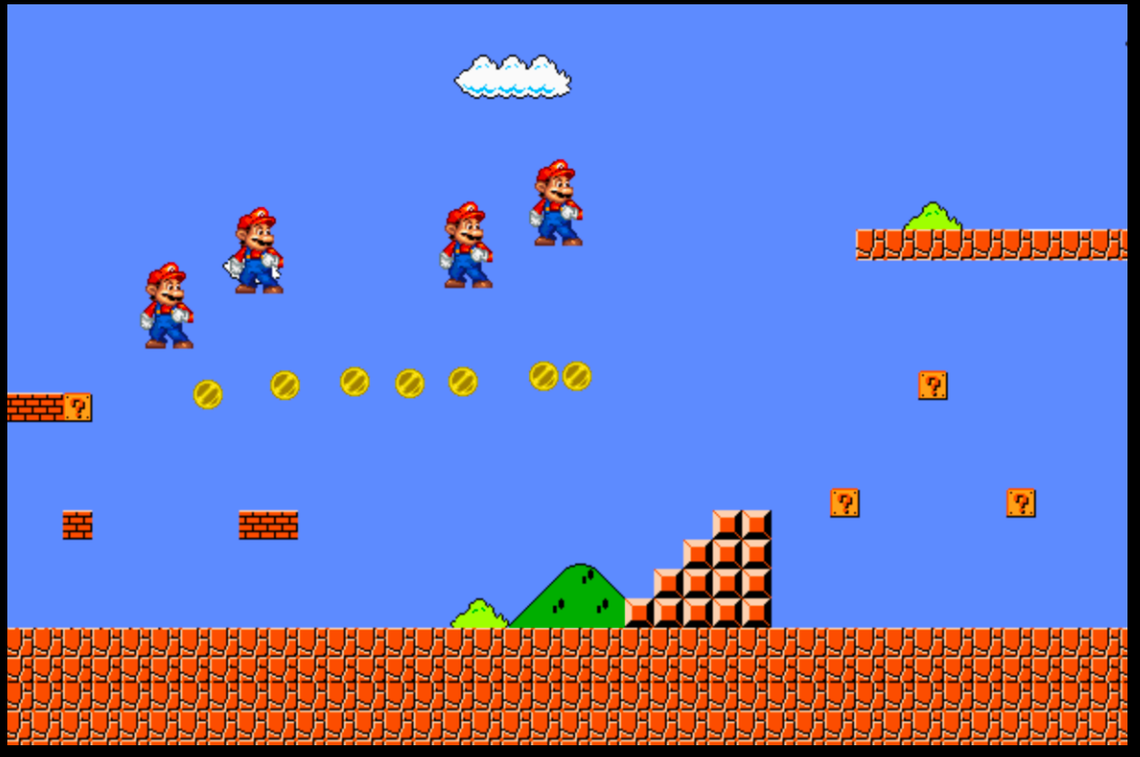
remove all the sprites

draw all the sprites in their new location

Multiple Groups

- `sprite4.py` shows a group for `mario` sprites and a separate group for `coin` sprites

- left mouse click places a coin
- right places a mario (who then starts to drop)



Other Classes

- `RenderUpdates` is a Group whose `update` method will return a list of all rects that were changed
  - You pass this to `display.update()` method and only those spaces get updated in the frame
- `OrderedUpdates` is a Group that draws the sprites in the order they were added
  - Controls for overlapping sprites

- `LayeredUpdates` is a Group that has layers
  - Sort of a group within the Group
- `GroupSingle` is a Group that can only contain a single sprite
  - If a new sprite is added, the old one is ejected

- `DirtySprite` can indicate if the image or rect has changed (and thus it might not need to be redrawn)
- `LayeredDirty` is a Group for DirtySprites. It will draw the sprites in the layer order, but only those sprites with the dirty attribute

Collisions

- A collision occurs when two sprites occupy the same space
  - We will talk more about this in a session to come
- The `pygame.sprite` module has methods to detect collisions

Sprite ➙ Sprite

- `pygame.sprite.collide_<something>()` returns `True` if two sprites overlap
  - There are various versions: `_rect` checks if the rects overlap
  - `_rect_ratio` multiplies the size of the rects by a floating point value before looking for overlap
  - `_circle` centers a circle on each sprite and sees if the circles overlap
    - `_circle_ratio` does what you expect
  - `_mask` checks if the mask of two sprites would overlap
    - The mask is a 1-bit per pixel array with the bit set for all pixels that are opaque in the sprite's image

Collision methods

- `pygame.sprite.spritecollide(sprite, group, dokill, collided=None)` returns a list of sprites in the group that collide
- If `dokill` is `True`, the collided sprites will be removed from the group
- `collided` is a *callback function* that will be used to calculate collisions
  - Yes, you are passing a function into a function
  - The intention is that you specify one of those functions we just talked about: collide_circle or collide_mask

```
#sprite5.py: Coins with right button, mario with left

mario_group = pygame.sprite.GroupSingle()
coins_group = pygame.sprite.Group()

while True:
    quit, click, click2 = check_events()
    if quit:
        break
    if click:
        mario = Mario(click, sprites_surf)
        mario_group.add(mario)
    if click2:
        coin = Coin(click2, sprites_surf)
        coins_group.add(coin)

    mario_group.update(window_size_y)
    if len(mario_group) > 0 and len(coins_group) > 0:
        pygame.sprite.spritecollide(mario_group.sprite, coins_group, dokill = True,
                                    collided=pygame.sprite.collide_circle)

    surface.blit(background, (0,0))
    mario_group.draw(surface)
    coins_group.draw(surface)
    pygame.display.flip()
```

Only a single Mario

Many coins

Must check for sprites in the group, else spritecollide fails

or _rect, _mask, etc

EXPERIMENT!!!

Group collision detection

- `spritecollide` checked if a single sprite collided with any of the sprites in a group
- `pygame.sprite.groupcollide()` tests for collisions between two groups
- `groupcollide(group1, group2, dokill1, dokill2, collided=None)`
  - Returns a dictionary with every sprite in group1 as a key
  - The value is a list of sprites in group2 that intersect
- Any of the `collide_*` functions can be passed in for collided

```
# sprite6.py: Group collision detection
mario_group = pygame.sprite.Group()
coins_group = pygame.sprite.Group()

while True:
    quit, click, click2 = check_events()
    if quit:
        break
    if click:
        mario = Mario(click, sprites_surf)
        mario_group.add(mario)
    if click2:
        coin = Coin(click2, sprites_surf)
        coins_group.add(coin)

    mario_group.update(window_size_y)
    pygame.sprite.groupcollide(mario_group, coins_group, False, True)

    surface.blit(background, (0,0))
    mario_group.draw(surface)
    coins_group.draw(surface)
    pygame.display.flip()
```

Many Marios

Handles empty groups, so no need to test beforehand

If there is a collision, kill the coin, not the Mario

animation

Animation

- To animate your sprites, show a quick sequence of images to the user
  - How quick? 30-72 frames per second or so
- You will be showing a different image each time around the game loop (i.e. different frames)
- So far, the game loop has just run as fast as it can, based on how long it takes to execute the code

File of many pictures

Load it and store surface

I've measured coordinates for all of the pictures

Called every time around the game loop

Draw the chosen picture at the updated rect

```
# draw_mario.py
class Mario(pygame.sprite.Sprite):

   def __init__(self, x, y, sprites_file):
        # As Marios' height varies, x, y will be measured from his lower left corner
        pygame.sprite.Sprite.__init__(self)
        ...

        self.image = pygame.image.load(sprites_file).convert()
        self.image.set_colorkey(self.image.get_at((0,0)))
        self.image_rects = [pygame.Rect(  4, 13, 29, 47),
                            pygame.Rect( 38, 15, 32, 45),
                            ....,
                            pygame.Rect(368, 15, 29, 45)
                           ]

    def update(self, right, jump):
        if self.y>=558:
        .... # Lots more code to figure out state and which image to draw
        if not jump and not right:
            x=0
            self.draw_index = 0
        ....
        self.rect = self.image_rects[self.draw_index].copy()
        self.rect.bottomleft = (self.x, self.y)
        self.blit_area = self.image_rects[self.draw_index]
        self.draw_index = x

    def draw(self, target_surface):
        target_surface.blit(self.image, self.rect, area=self.blit_area)
```

Time and the Game Loop

- If the game loop takes different amounts of time to execute, our animations will not be smooth or fluid
  - If one frame goes takes 1/90 sec to compute and the next takes 1/30 (more computation, so slower), then the animation will be jerky
- Two approaches:
  1. Slow the game loop down to a consistent number
  2. Measure the time of the game loop and use that to compute the sprite updates

Approach 1: Slower loop

- `pygame.time.Clock()` is a measurement object
- It has a `tick` method, that will delay until a certain framerate has passed

```
clock = pygame.time.Clock()
while True:  # beginning of game loop
    clock.tick(30)  # will delay until 1/30th of a second
                    # since the last call to this method
```

Approach 2: Measure

- The `Clock` object can also be used to measure the time since the last call
  - If clock.tick is called without a parameter (or with zero), it will not delay
    - But, it will return number of milliseconds since last time

```
clock = pygame.time.Clock()
while not done:  # beginning of game loop
    delta = clock.tick()  # no or zero parameter means no
                          # forced fps delay
    *user_inputs, done = get_inputs()
    game_state = update_game_state(user_inputs, delta)
    render_game(game_state, delta)
```

delta value (number of milliseconds for the last loop)
eventually gets to the Sprite update methods

```
class Mario(pygame.sprite.Sprite):
....
    def update(self, left, right, jump, delta):
        if left:
            self.x -= self.velocity * delta
        ....
```

randomness

Randomness

- Games often generate different outcomes for some event, even if you play the game again
  - Ex: treasure in a chest, hit points of the boss
- Problem: Computers are very discrete systems
- They are designed to do the same thing with the same input
  - Where can randomness come from?

Pseudo-randomness

- Linear Congruential Generator (LCG) is an algorithm, which when performed repeatedly, seem to generate a random sequence of numbers

$$X_{n+1} = (a * X_n + c) \bmod m$$

ex: $a = 1664525$, $c = 1013904223$, $m = 2^{32}$

- Good enough for some "random" uses
  - Pseudo-Random Number Generator (PRNG)

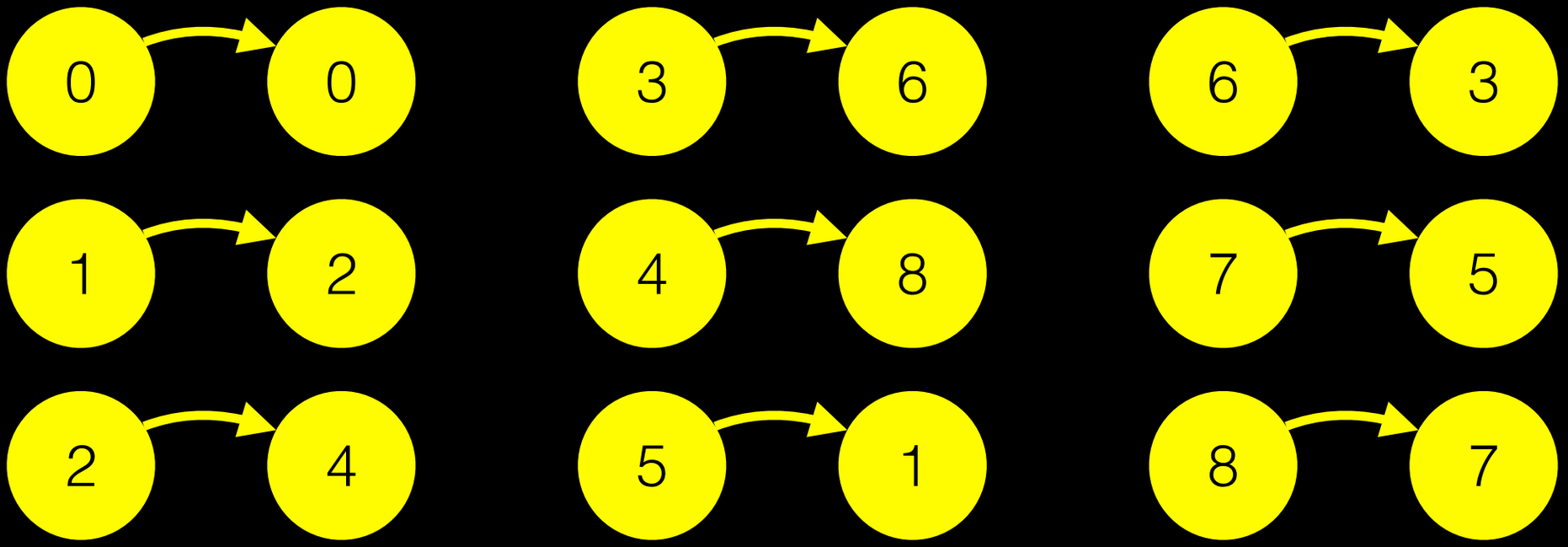
LCG Cycles

- As the output of an LCG has a specific range (because of the modulo operator), it will eventually repeat
  - The *cycle* is the number of times you can call the LCG before it repeats

$X_{n+1} = (a * X_n + c) \bmod m$

- For a = 2, c = 0, m = 9 there is a maximum cycle of 9 (you only have 9 possible outputs, 0-8)

- But, if you check every possible input, you'll realize your cycles are smaller



PRNG

- A good PRNG will have carefully chosen constants to ensure the longest possible cycle
- Also, other sources of "randomness" are usually sampled and mixed in
  - inter-keypress times, mouse locations, etc

Python P-Randomness

- Library in module random provides pseudo-random numbers of fairly good form

```
import random
```

`random.random()` → float [0.0, 1.0)

`random.randrange(start, stop)` → integer [start, stop)

`random.shuffle(seq)` → shuffles the sequence seq in place

`random.seed(val)` → initializes to known starting point

Beware!

- For some uses, special care required
  - Cryptography / Security
  - Lotteries / Gambling
- Use Python's `secrets` module for these
  - And, even then, be cautious!

```
import random

def roll_3_dice():
    total = 0
    for _ in range(3):
        total += random.randrange(1,7)
    return total
```

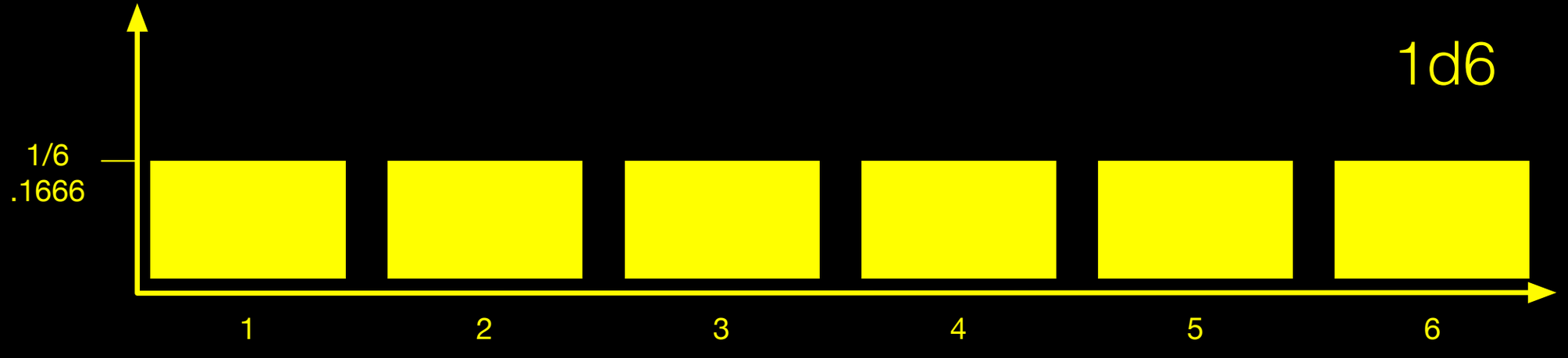
Don't care about the loop index

Each call returns either 1,2,3,4,5,6

random distributions

Distribution

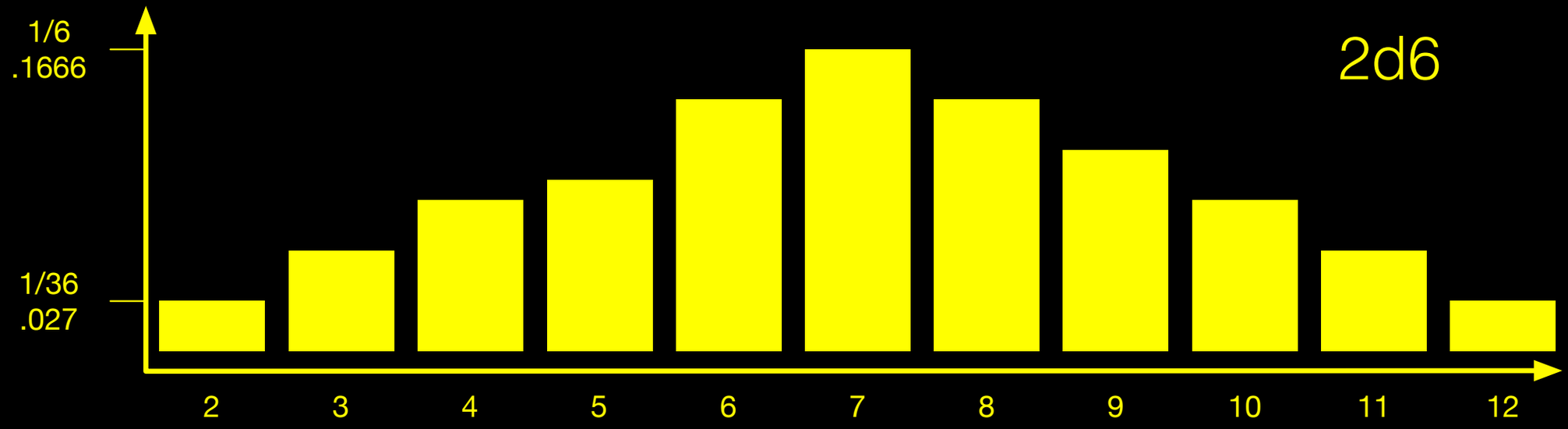
- The *distribution* of a PRNG or code using a PRNG refers to the chance of attaining each value
- For instance, rolling a single 6-sided die (or doing `random.randrange(1,7)` gives 1/6 probability for each of 6 possible outcomes ➞ uniform distribution



The bigger range, the lower the probability of any one result

Adding Randomness

- Adding several PRNGs of uniform distribution does not result in a uniform distribution
- Here is random.randrange(1,7) + random.randrange(1,7), a range from 2-12



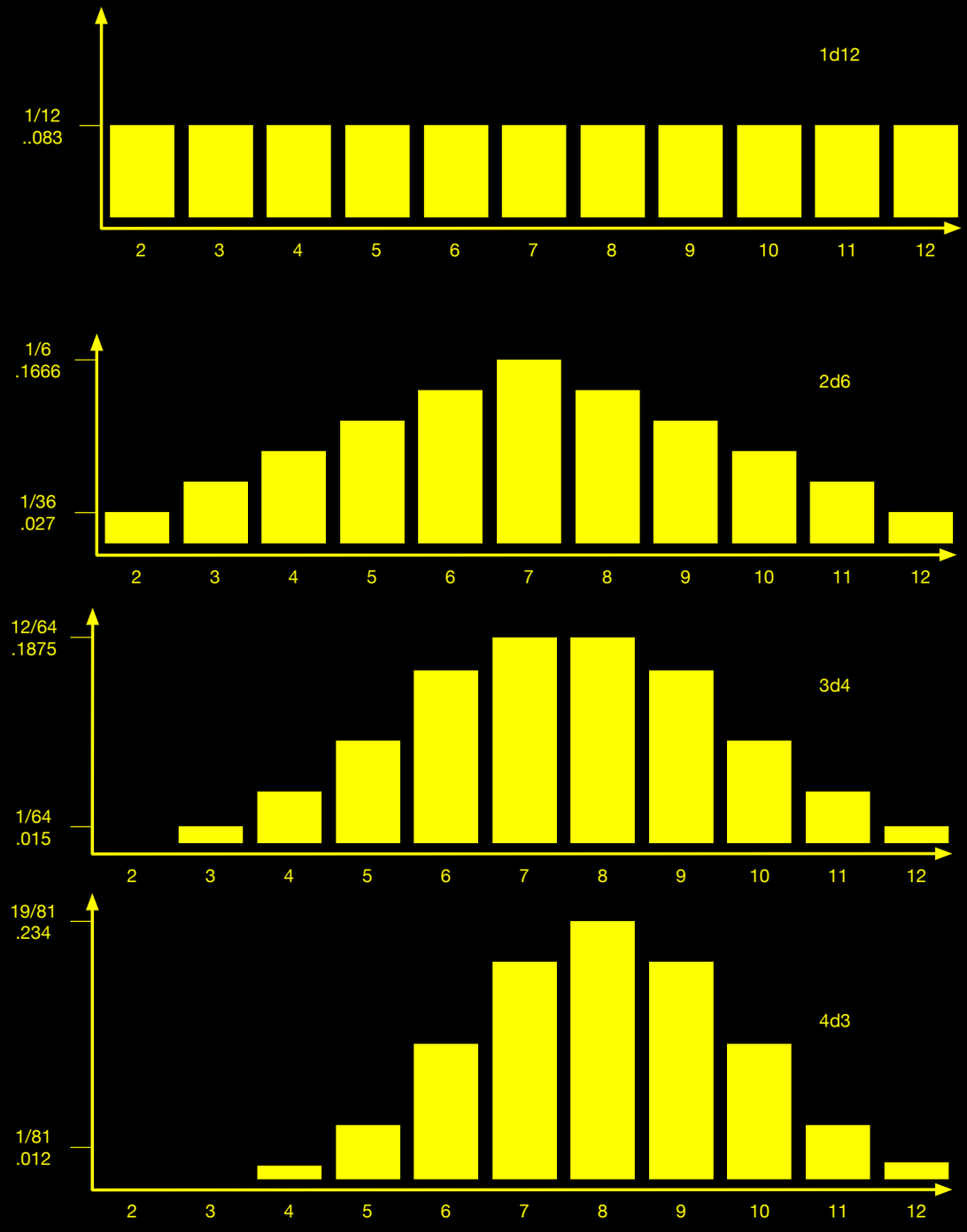
Calculating the Distribution

- What are all possible outcomes?
  - 2d6 → (1,1), (1,2), ... (1,6), (2,1), (2,2)... (6,6)
- How many add to 2? To 3?
- Code on next page gives this answer

```
2 → 1, 0.028
3 → 2, 0.056
4 → 3, 0.083
5 → 4, 0.11
6 → 5, 0.14
7 → 6, 0.17
8 → 5, 0.14
9 → 4, 0.11
10 → 3, 0.083
11 → 2, 0.056
12 → 1, 0.028
```

```
# Calculating the distribution
# What are all possible rolls?
rolls = []
for i in range(1,7):
    for j in range(1,7):
        rolls.append((i,j))
# rolls = [(1, 1), (1, 2), (1, 3),...(6, 4), (6, 5), (6, 6)]
# Add up the outcome for each roll
outcomes = [sum(x) for x in rolls]
# outcomes = [2, 3, 4, 5, 6, 7, 3, 4, 5...9, 10, 11, 12]
# Count how many of each outcome
import collections
c = collections.Counter(outcomes)
# How many total rolls? len(rolls)
for i in range(min(c),max(c)+1):
    print(f'{i} → {c[i]}, {c[i]/len(rolls):0.2}')
```

- Several distributions in the range of 12
- Notice, distribution narrows as more, lower valued, dice are used
- Also, distribution shifts from left to right

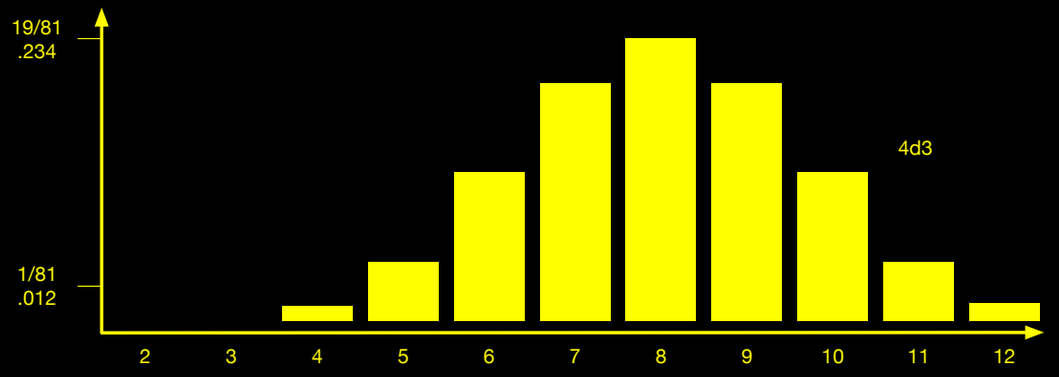


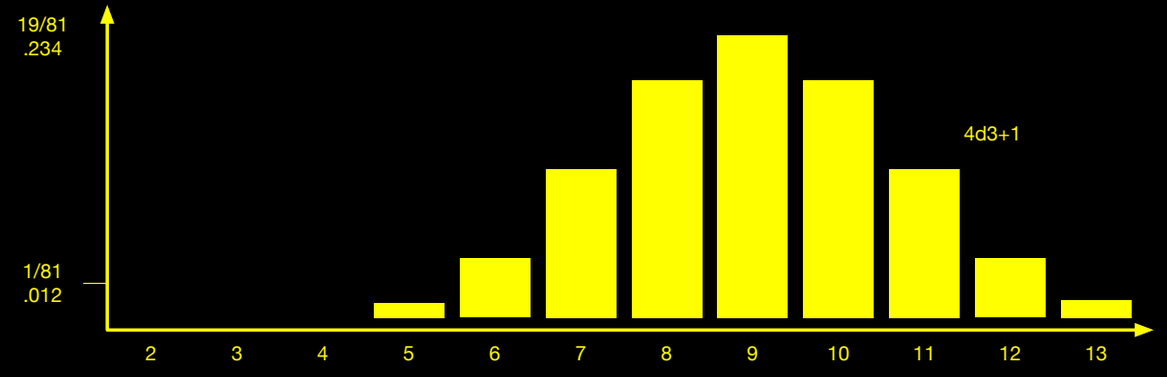
Designed Distributions

- What if we wanted to have some other distribution, to model some effect in our game?
- How shall we modify the given distributions?
- BTW: I'm using "dice" and "sides" to make these explorations a bit more concrete
  - Obviously, we can use random to get values from non-dice-related ranges

Shifting the distribution

- You can easily shift the distribution by adding (or subtracting) a value to the total
- You will adjust the average, but not change the shape of the distribution





```
# Code aside

def roll_dice(number, sides):
    ''' Return the sum of N dice rolled
        Each dice has sides possible values
    '''
    sum = 0
    for _ in range(number):
        sum += random.randrange(1,sides+1)
    return sum
```

Asymmetric Distributions

- Often in our games, we want to shift the distribution a bit, often to favor the player somehow (who wants to play a hero with strength of 5?)
- How can we adjust the distribution so it is no longer symmetric (as all so far have been)?
- How can I get a higher chance of values at the upper ranges?
- There are several different ways

Dropping the lowest roll

- Roll twice (or more), pick the higher roll
  - Now, the only way to get the lowest value is to roll it twice

```
def drop_lowest_roll(number, sides):
    roll1 = roll_dice(number,sides)
    roll2 = roll_dice(number,sides)
    return max(roll1, roll2)
```

Drop the lowest dice

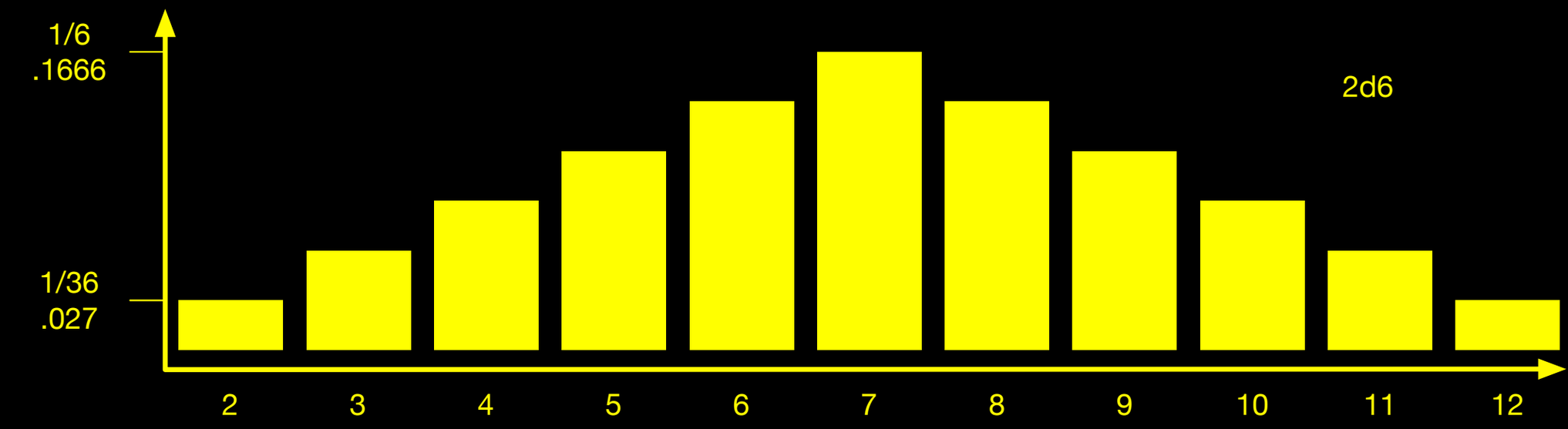
- A bit more asymmetry results if we roll an extra die and drop the lowest value

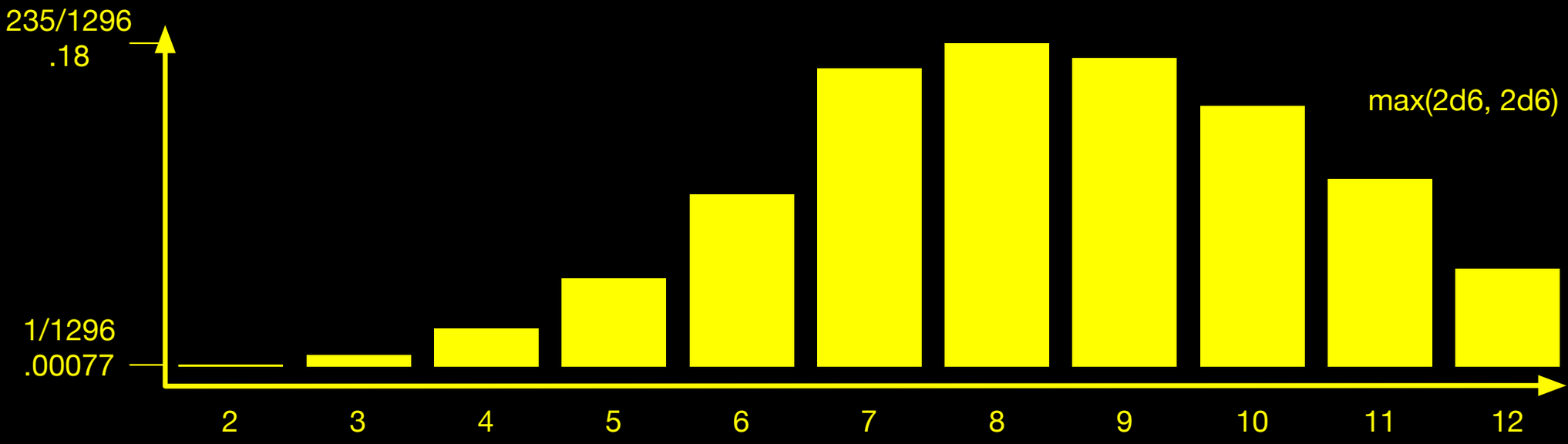
```
def drop_lowest_die(number, sides):
    rolls = []
    for _ in range(number+1):
        rolls.append(roll_dice(1, sides))
    rolls.sort()
    return sum(rolls[1:])
```

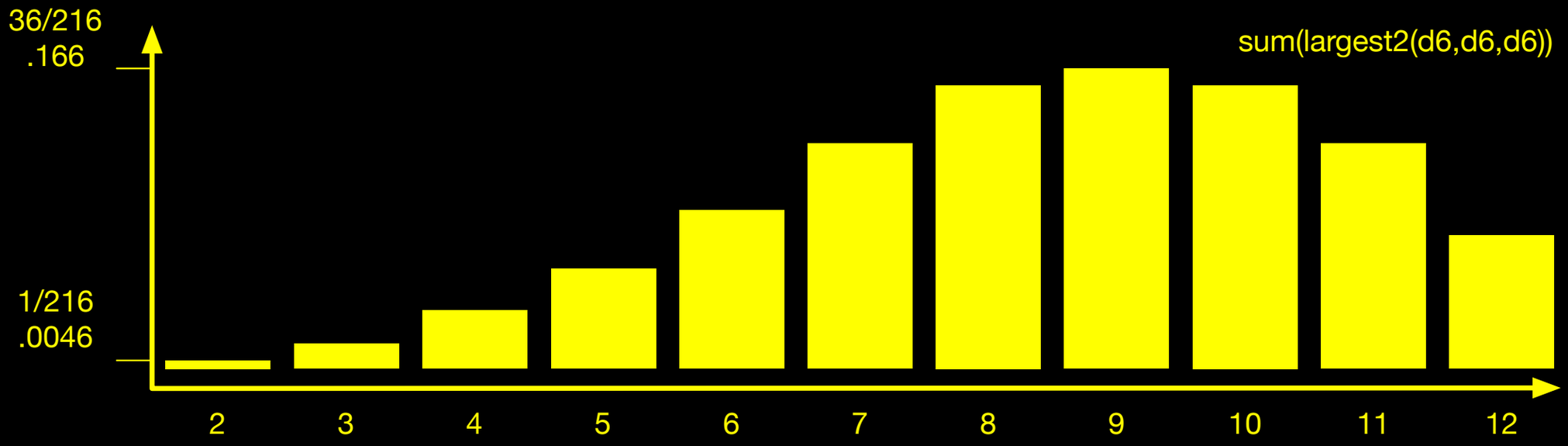
Reroll the lowest

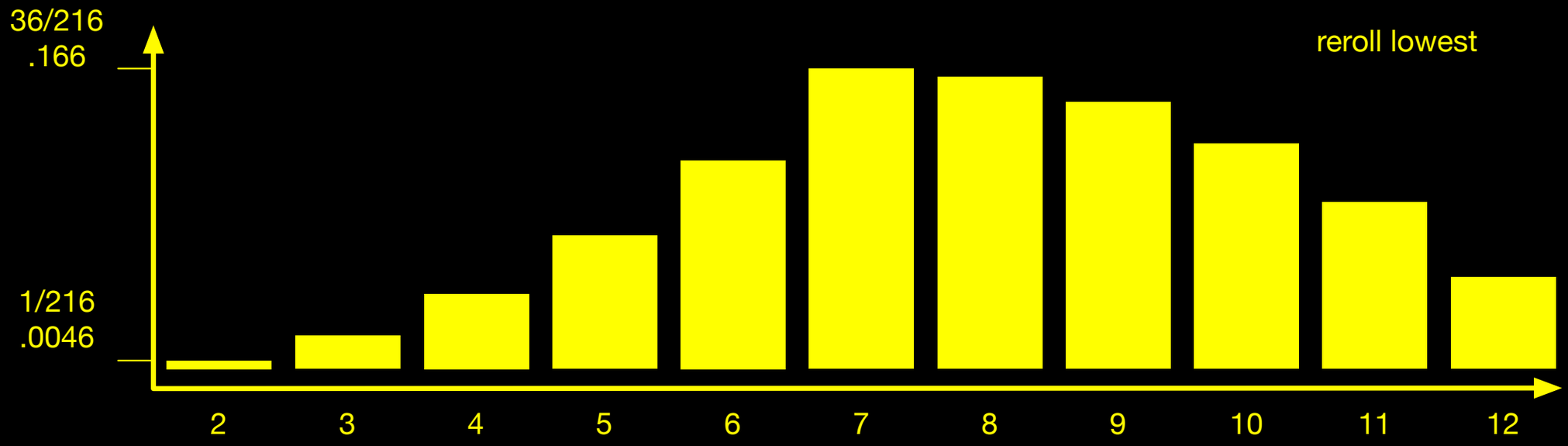
- A more gentle, less severe, asymmetry happens if you roll two dice and then reroll whichever is lowest

```
def reroll_lowest_die(number, sides):
    rolls = []
    for _ in range(number):
        rolls.append(roll_dice(1, sides))
    rolls.sort()
    rolls[0] = roll_dice(1, sides)
    return sum(rolls)
```

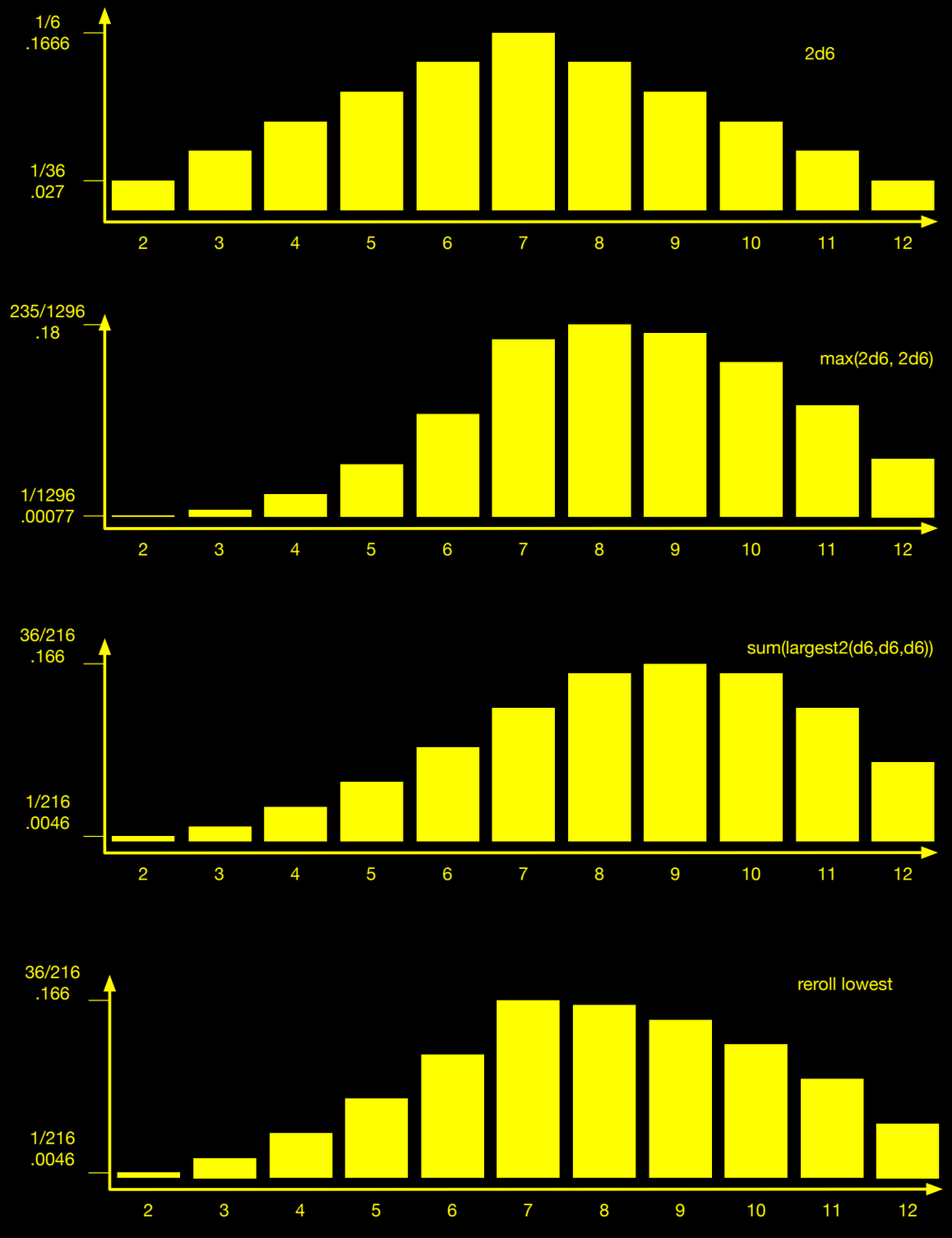

1/6
.1666
1/36
.027
2d6
2
3
4
5
6
7
8
9
10
11
12


235/1296
.18
1/1296
.00077
max(2d6, 2d6)
2
3
4
5
6
7
8
9
10
11
12


36/216
.166
1/216
.0046
sum(largest2(d6,d6,d6))
2
3
4
5
6
7
8
9
10
11
12


36/216
.166
1/216
.0046
reroll lowest
2
3
4
5
6
7
8
9
10
11
12

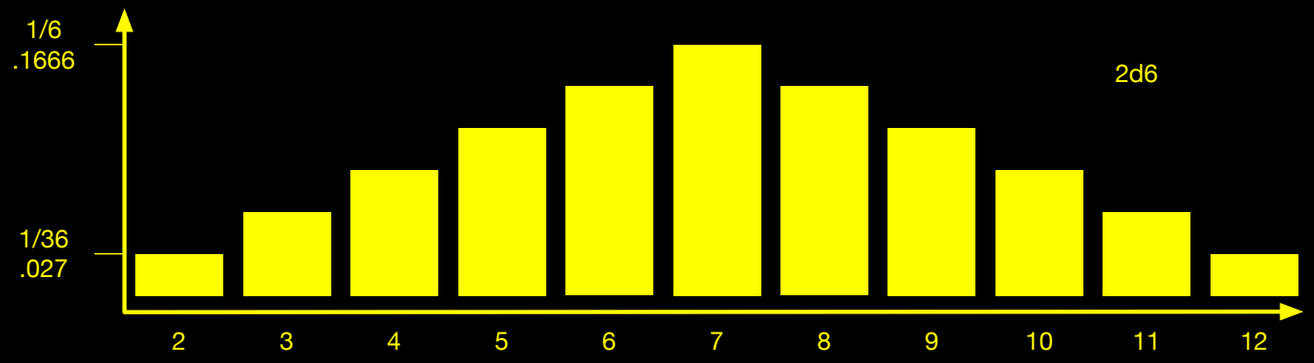
All 4 together

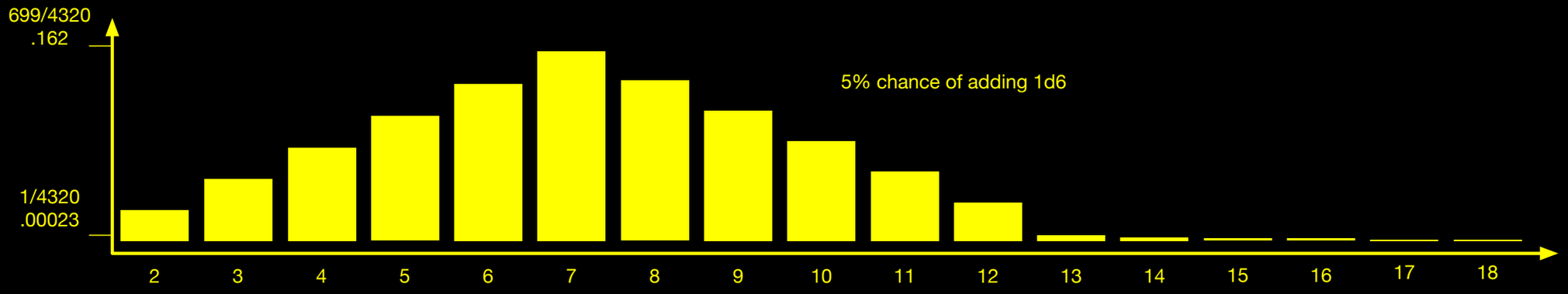


"Critical Hit"

- Critical Hit: In a small percentage (5%?) of cases, extra value is added (perhaps by another die roll)
  - Has huge psychological advantages
  - Doesn't actually move the distribution all that much

```
def roll_with_critical(number, sides, crit):
    roll = roll_dice(number, sides)
    if random.random < crit:
        roll += roll_dice(1, sides)
    return roll
```

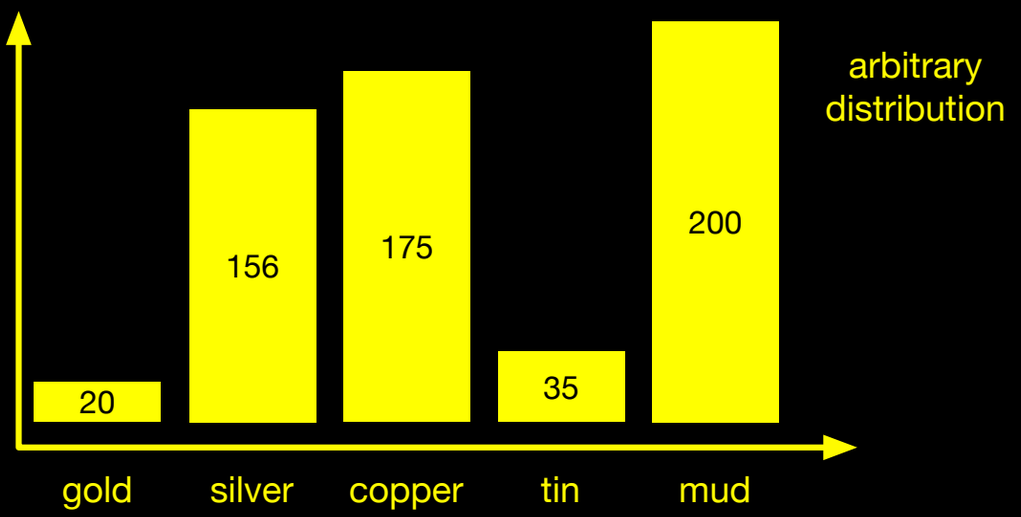

1/6
.1666
1/36
.027
2d6
2
3
4
5
6
7
8
9
10
11
12


699/4320
.162
1/4320
.00023
5% chance of adding 1d6
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

Arbitrary Distributions

- You can, of course, design your own distribution and then generate random numbers against it
  - The distribution may even be a result of in-game, run-time effects (i.e. the current economy, how happy the king is, etc)
- Basically, you add up the desired probability for each of the possible outcomes
- Then, generate a random number up to that sum
- Create a result by subtracting out the probability for each outcome until you get to zero

```
# Example of Arbitrary Distribution
total = sum(20,156,175,35,200)
roll = randrange(0,total)
if roll < 20:
    print("Gold!!")
elif roll < 20 + 156:
    print("Silver!")
elif roll < 20 + 156 + 175:
    print("Copper")
elif roll < 20 + 156 + 175 + 35:
    print("tin")
else:
    print("mud")
```



```
# Better arbitrary distribution result maker
dist = [ (20, "Gold!"), (156, "Silver"),
         (175,"Copper"), (35, "Tin"), (200, "Mud") ]

total = sum([x[0] for x in dist])
roll = random.randrange(total)
for (weight, result) in dist:
    if roll < weight:
        win = result
        break
    roll -= weight
print(f'Congratulations!  You have won some {win}')
```

Randomness Conclusions

- Python makes good randomness easy (random module)
- Distributions matter in your game
- Use the number of rolls to control the variance
  - The distribution is wider for lower number of rolls
- Use the die-size to control the scale
  - Dice with more sides give you bigger scale

Randomness Conclusions (2)

- Offsets can slide the distribution back and forth to wherever you might want it
- Asymmetrical distributions are often useful
  - Drop lowest die, roll twice (thrice?) and take the max, re-roll the lowest, etc
  - Critical hits often feel great (see that cool special effect?), but may not change the distribution very much
- Arbitrary distributions are easy

What did you learn today?

- Fonts
- Sprites
- Randomness