Starting chapter 6

Image processing introduction

- Digital image data are stored one of two ways
  - Vector data – points, lines, polygons, …
    - Efficient way to store data; facilitates analysis (and plotting)
  - Raster data are more common though – rows/columns of picture elements (pixels), each a particular color
    - Most common way to capture data; easy to display on-screen
- Text's `cImage` module processes raster data
  - Designed to work with `.gif` and `.ppm` formats only
    - Can install a library for `.jpg` format, but not available in lab
  - Chapter 6 uses objects of the module's `Pixel`, `FileImage`, `EmptyImage` and `ImageWin` classes

A `Pixel` class

- A way to manage the color of one pixel
- A `color` = *amounts* of (`red`, `green`, `blue`)
  - When coded by the RGB color model
  - Range of each part: 0-255
    - So 256×256×256 = 16,777,216 possible colors on-screen (but alas, `.gif` format only stores a *palette* of 256 of them!)

```
whitePixel = cImage.Pixel(255,255,255)
blackPixel = cImage.Pixel(0,0,0)  # opposite of paint
purplePixel = cImage.Pixel(255,0,255)
yellowPixel = cImage.Pixel(255,255,0) # surprise!
```

- Methods: `getRed()`, `setBlue(value)`, …

Image classes in `cImage`: `EmptyImage` and `FileImage`

- Technically both subclasses of `AbstractImage` – so objects have exactly the same features
  - Create new: `cImage.EmptyImage(cols, rows)`
  - Or use existing: `cImage.FileImage(filename)`
- Really just a way to manage a set of pixels, organized by rows and columns
  - `x` denotes the column – leftmost `x` is 0
  - `y` denotes the row – topmost `y` is 0
- Methods: `getWidth()`, `getHeight()`, `getPixel(x, y)`, `setPixel(x, y, pixel)`, `save(filename)`, … and `draw(window)`

`ImageWin` class

- A window frame that displays itself on-screen
  - And lets an image draw (itself) inside

```
window = cImage.ImageWin(title, width,height)
image.draw(window)
```

- Mostly just used to hold images, but also has some methods of its own
  - e.g., `getMouse()` – returns `(x,y)` tuple where mouse is clicked (in window, not necessarily same as image)
  - `exitOnClick()` – closes window and exits program on mouse click (like `turtle.screen` feature)

Try it!

Simple image processing ideas

- Basic approach creates new image in 3 steps *for each pixel* in existing image:
  1. Get the existing color components (r, g, b)
  2. Build a new pixel – usually a function of (r, g, b)
  3. Insert new pixel into same (or related) position of new image
- Notice what "for each pixel" implies
  - Usually processing involves nested loops:

```
for row in range(height):
    for col in range(width):
```

Negative Images & Grayscale

- Negative images – "flip" each pixel color

```
for row in range(height):
    for col in range(width):
        # get r, g, b from old image here
        negPixel = Pixel(255-r,255-g,255-b)
        newImage.setPixel(col,row,negPixel)
```

  – Listings 6.1 and 6.2 – negimage.py
- Grayscale similar (Listings 6.3 and 6.4):

```
# ... as above through get r, g, b
        avg = (r + g + b) // 3
        grayPixel = Pixel(avg,avg,avg)
```

  – Listings 6.3 and 6.4 – grayimage.py

Abstraction by function parameter

- Hmm… same except `newpixel = f(oldpixel)`
- General solution – *pass a function*:

```
def pixelMapper(oldImage, rgbFunction):
    # nested loops – for each oldPixel in oldImage:
        newPixel = rgbFunction(oldPixel)
    # returns newImage at end
```

- Now just pass function name for desired effect

```
negImage = pixelMapper(oldImage, negPixel)
grayImage = pixelMapper(oldImage,grayPixel)
```

- Listings 6.5 and 6.6 – genmap.py

Using functions to write programs

- Another structured programming idea: modularity
- Can directly translate an algorithm – e.g.,

```
data = getData()
results = process(data)
showResults(results)
```

- In turn, the function `process()` might include:

```
intermediateResult = calculate()
```

  to let a function named `calculate` do part of the work

  – And so on …

Note: parameters are *copies*

- e.g., `def foo(x):`
  `x = 5` # changes *copy* of the value passed
- So what does the following code print?

  ```
  a = 1
  foo(a)
  print(a)
  ```

  – Answer: `1`
- Applies to all *immutable* objects, inc. strings

  ```
  s = "APPLE"
  anyMethod(s)
  print(s)    # prints APPLE
  ```

But references *are* references

- A reference is used to send messages to an object
- So original object can change if it is mutable
- e.g., `def foo(myTurtle):`

```
          myTurtle.forward(50)
```

  `#` actually moves the turtle
- Copy of reference is just as useful as the original
  - Whereas functions cannot change a reference, they can change the original object by using the reference
- So: be careful passing mutable object references

Scope/duration of variable names

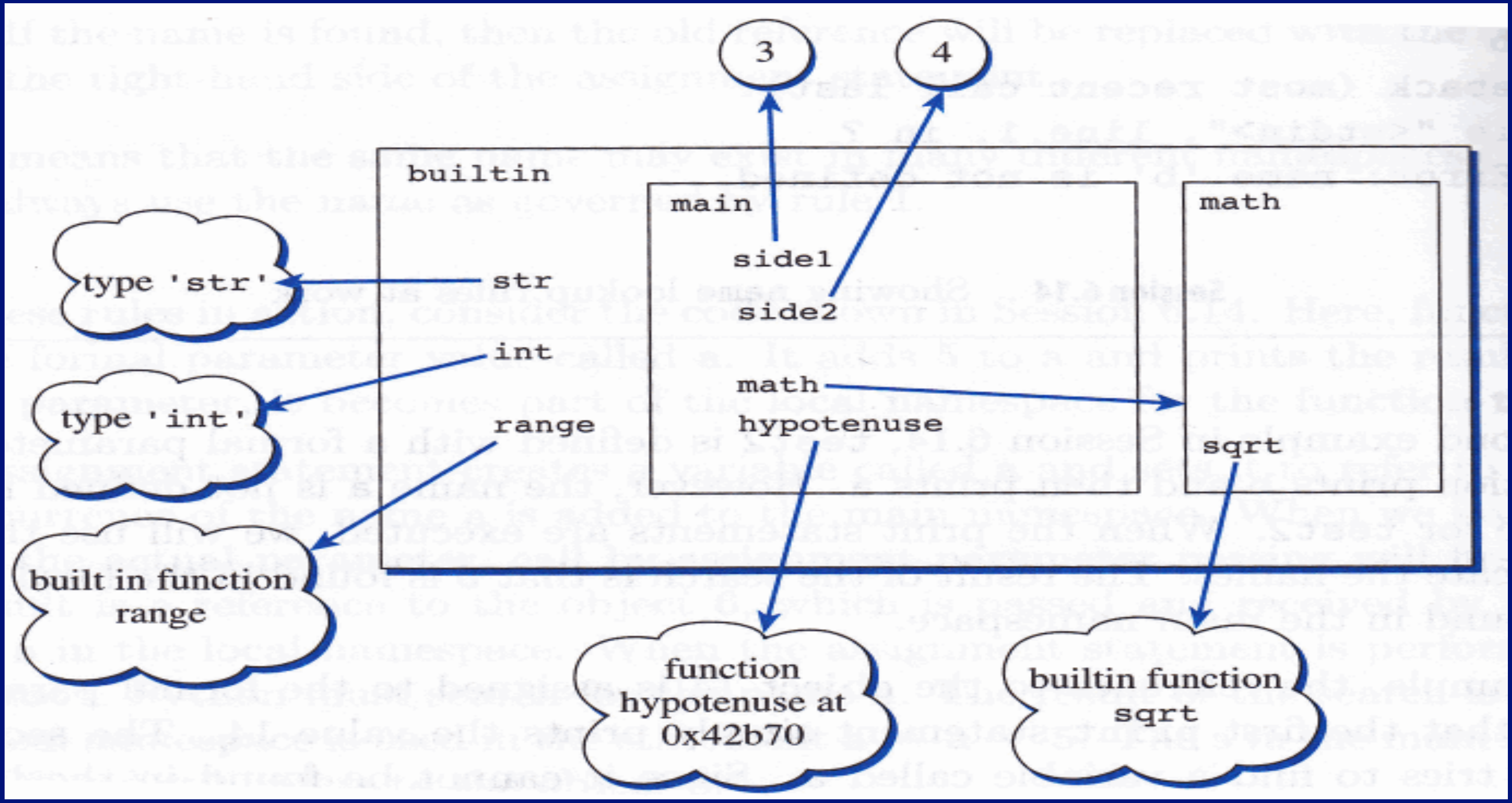
- Depends on namespace *where variable is created*
  - Rules differ by language – following are Python rules
- Global variables (created outside any function):
  - Duration ("lifetime"): until program exits
  - Scope: available everywhere after first creation, even inside functions that follow – but can be hidden inside a function by a variable that has the same name
- Local variables created in a function (including the parameters that get created as copies):
  - Duration: as long as function is being executed
  - Scope: available after creation, but just in the function

Try it!

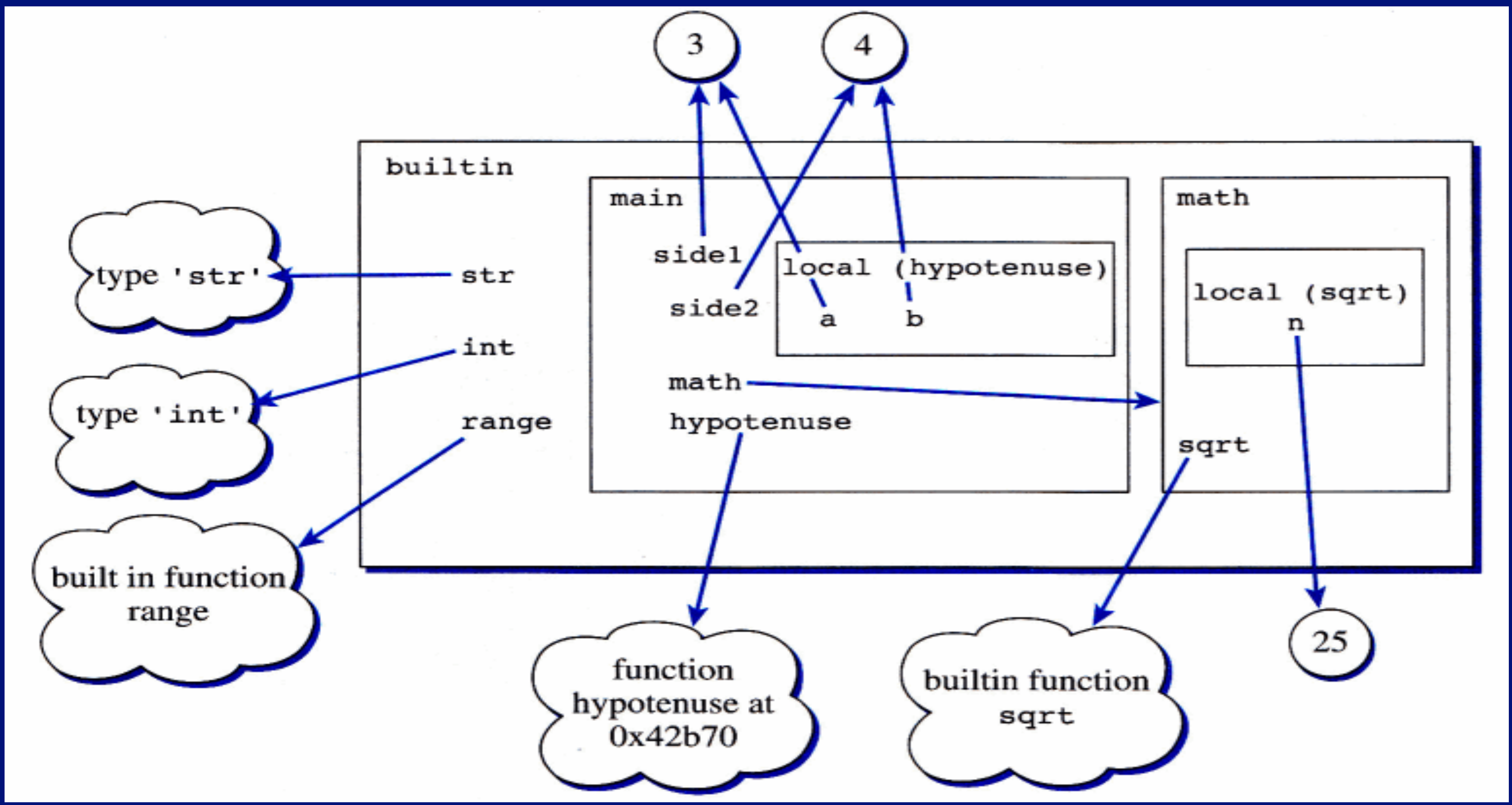
Namespaces

- Def: the names available for a program to use – at a particular point in the program's execution
- Every Python program starts with two namespaces
  - `__builtins__` – built-in namespace includes system-defined names of often-used functions and types
    - Try: `>>> dir(__builtins__) #` to get a list
  - `__main__` – your program's namespace (starts empty)
    - Try: `>>> dir() #` (with no arguments) – boring at first
    - Populate it: create variables, define functions, import modules
- A function/module has its own *local* namespace

Example namespaces (Figure 6.12)

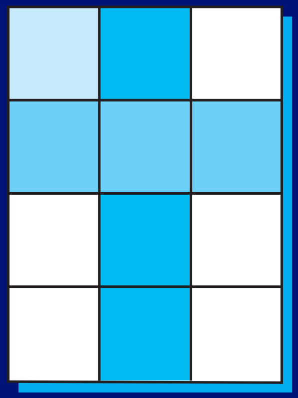


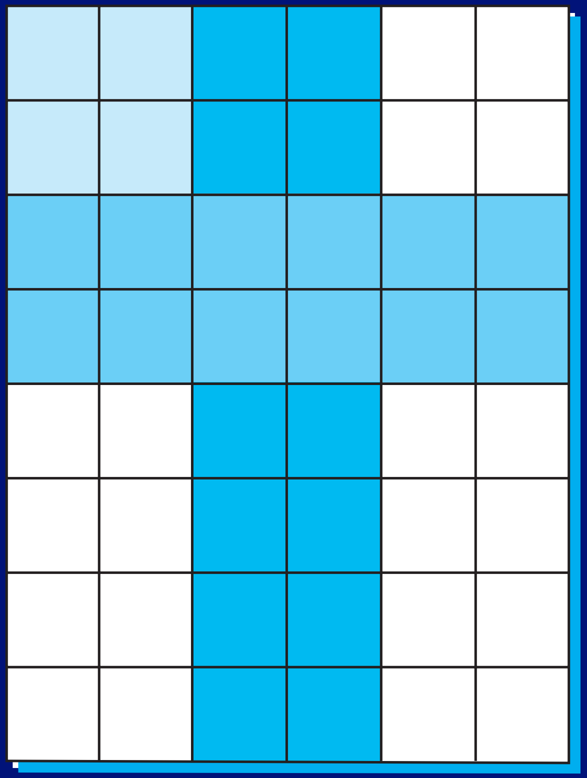
Local namespaces (Figure 6.13)



Doubling the size of an image

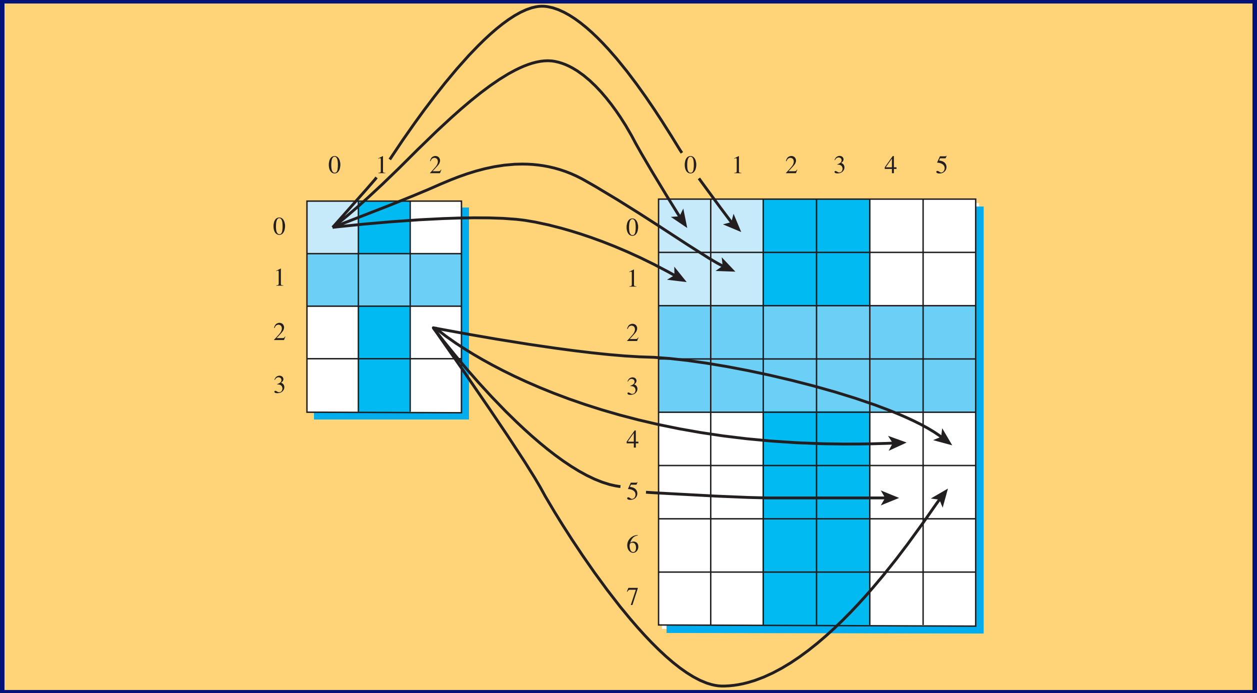
- Each old pixel → 4 new pixels





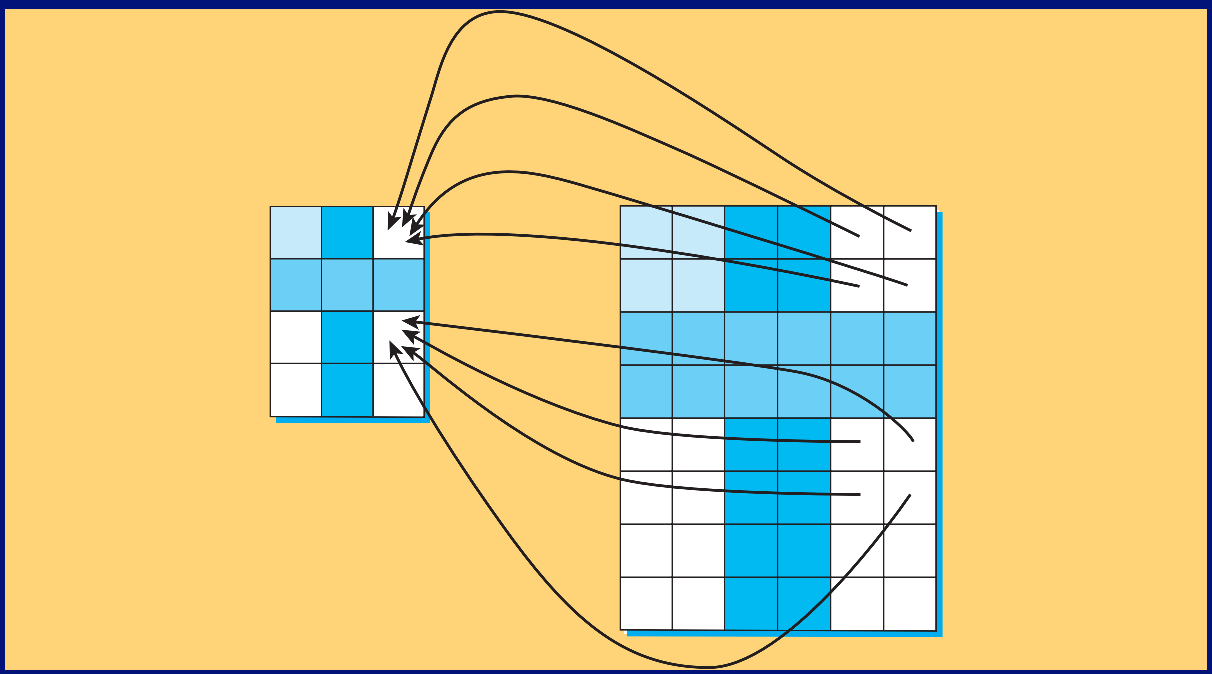
Doubling – one way to do it

- Listing 6.8 – Loop through old image rows/columns: for each pixel set 4 new image pixels



Doubling – another way to do it

- Listing 6.9 – Loop through new image rows/ columns: set each pixel to associated pixel in old image



Results in both cases look “grainy” or “blocky” – because not adding detail. Can “smooth” based on colors of pixel neighbors.

Flipping or rotating an image

- Both techniques involve moving pixels around
- Flipping on vertical axis, for example: →←

```
maxp = width-1 # max pixel (new has same width and height)
for row in range(height):
  for col in range(width):
    oldPixel = oldImage.getPixel(maxp-col,row)
    newImage.setPixel(col,row,oldPixel)
```

- Rotating does make the new image a different size than the old one (unless rotating by 180°)

Edge detection – more complex

- An edge is where neighboring pixels differ dramatically
- Classic way uses a “kernel” (a.k.a., mask or filter) for each direction, x and y

-1	0	1
-2	0	2
-1	0	1
X_Mask		

1	2	1
0	0	0
-1	-2	-1
Y_Mask		

- Process by “convolution” – combine intensities of neighboring pixels (multiply by mask values and sum over all neighbors, for each mask)

More edge detection

- Can represent a mask as a list of lists
  - e.g., `xMask = [ [-1,-2,-1],[0,0,0],[1,2,1] ]`
  - Listing 6.11 returns convolution sum for one mask / one pixel
- Main edge-detect function (Listing 6.12) creates gray scale image, then loops once for each pixel to create new image
  - Calls convolve function for each mask – gets `gx`, `gy`
  - Calculates `g` = square root of $(gx^2 + gy^2)$
  - Sets pixel color black if `g` > threshold value (recommended value is 175) – otherwise pixel set white
- Alternatively, can skip gray scale step and set pixels red, green or blue based on separate convolutions (rgbdetectedges.py)

Next

More Python and programming topics