Photo#: A Photo Editing Language

Language Reference Manual

Manager	Language	Architect	Architect	Testing
Kevin Chu	Michael Block	Marie Matthews	Jancy Capellan	Julio Melchor
kc2991	mb4239	mpm2183	jc4491	jjm2226

Table of Contents

I. Introduction

Photo# is a language built for simple image editing, processing, and filtering tasks and operations. These use cases include brightness and contrast adjustments, cropping, rotation, and simple color or black/white filtering.

Motivation

Photo# abstracts away the low-level nature of image processing algorithms, **abstracting away the doubly-nested for loop** in particular. The language includes abstractions in the form of **keyword function operations** (e.g. brighten, saturate, hue, rotate) on images.

Photo# is **Pythonic** and written for user-friendliness, as to reflect the user-friendly nature of image editing applications our end user will be familiar with.

The language aims to enable an in-line scripted image editing experience that approaches the ease of a front-end experience such as Instagram. Photo# provides a considerably more **user-friendly composable iterator** and **implicit variables** for iterating through Pixel data types in Image composite data structures.

II. Lexical Conventions

We identify six different types of tokens: identifiers, keywords, literals, constants, operators, and separators.

Photo# accepts an alphanumeric set of characters `a-z A-Z 0-9` as well as special characters such as `,<>_();$:%[]#?'&{}"^!*/|-\~+.`—in other words, the entire ASCII character set.

Comments

```
'''
This is a comment
'''

'''
Multi-line Comments are enclosed by two sets of three quotation marks
Comments cannot be nested
'''

# Single line comments are indicated with hashtag
```

Identifiers

Identifiers must be unique names for variables, functions, and such identified entities. Identifier naming follows these rules:

1. Valid identifiers may incorporate uppercase and lowercase letters, digits 0-9, dashes, and underscores as long as they are not a keyword.
2. As such, identifiers may not incorporate special characters outside of the alphanumerics, dashes, and underscores.
3. Identifiers may begin with a letter or digit, but not an underscore nor a dash.

```
'''
Valid identifier names
'''

Vacation01
img-4115

'''
Invalid identifier names
'''

pixel
goodmorning&
-img_007
_edit
```

Keywords

Keywords may not be used as identifiers. The following are reserved keywords in Photo#:

```
if
else
for
while
return
true
false
null
import
save
print
in
```

Literal type specifiers are also reserved keywords:

```
int
float
string
bool
```

```
arr

Pixel
Pixel.red
Pixel.green
Pixel.blue
red
green
blue
Image
Album
Caption
Gradient
```

Implicit variables

Keyword iterators default to the variable at the highest level of nesting, such that one may perform the following implicit variable operation:

```
'''
Brightens every pixel in Image
'''

for pixel in img:
      brighten(10)

'''
pixel.brighten() is implicit
'''
```

Constants

Integer constants

Integer literals consist of sequences of numeric digits [0-9]: `1234567890`

Character constants

Character literals consist of sequences of ASCII-defined characters.

For example, the alphabetic letters [a-Z], lowercase or uppercase, enclosed by single quotes: `abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ`

String constants

String literals are arrays of characters enclosed by double quotes. Strings will terminate with the null byte `'\0'` for the compiler to mark the end of a string.

Escape characters

Non-printable characters can be represented by the following escape sequences:

```
'''
Newline
'''
\n

'''
Tab
'''
\t

'''
Single quote
'''
\'

Double quote
'''
\”

'''
Whitespace
'''
' '
```

Primitive Declaration:

```
'''
Declare an integer
'''
int x = 2019;

'''
Declare a float
'''
float y = 21.2

'''
Declare a boolean
'''
bool is_true = true
bool is_false = false

'''
Declare a character
'''
char first_letter = 'a'

'''
Declare a string
'''
string school = "Columbia"
```

III. Data Types

Photo# supports the following data types, including composite data types helpful for the context of image editing.

Primitive Data Types

int

Stores a non-decimal number in 32 bits.

```
int i = 5
```

float

Stores a decimal number in 32 bits.

```
float f = 5.0
```

char

Stores a character value equivalent to any value in the ASCII table in 8 bits.

```
char c = 'a'
```

bool

Stores a value of true or false in 1 bit.

```
bool b = true
```

string

A sequence of ASCII value characters, denoted by double quotes “”.

```
string s = “hello”
```

array

A fixed size sequence of any type. All values in an array must be of the same type. Can be created in any of the following ways:

```
'''
Declare array1 of size 5, type string
Then set elements individually
'''
arr array1 = string[5]
array1[0] = “a”
array1[1] = “b”
array1[2] = “c”
array1[3] = “d”
array1[4] = “e”

'''
Declare array2, type string
Set elements on declaration -- size will be 5
'''
arr array2 = string[“a”, “b”, “c”, “d”, “e”]
```

Custom Composite Data Types

Pixel

The Pixel data type contains the values { red, green, blue } for image RGB color values. (x, y coordinates are implied by the Image array.)

RGB color values range from 0-255.

Users can also introduce defined color keywords per front-end design standards for ease of working with user-defined color values.

```
'''
Declare Pixel p with values:
red = 98
green = 168
blue = 229
'''
Pixel p = Pixel(98, 168, 229)

'''
Custom color values can be defined as global reference Pixels
'''
```

```
Pixel white = Pixel(255, 255, 255)
Pixel black = Pixel(0, 0, 0)
Pixel yellow = Pixel(255, 255, 0)
Pixel honeydew = Pixel(245, 255, 240)
Pixel azure = Pixel(240, 255, 255)

'''
red, green, blue are keywords referring to the color channels of a Pixel,
therefore, red, green and blue, could be user-defined as follows:
'''

Pixel user_red = Pixel(255, 0, 0)
Pixel user_green = Pixel(0, 255, 0)
Pixel user_blue = Pixel(0, 0, 255)
```

Image

The Image data type is a stack whose initial element is a 2D array of Pixels containing the data of the Image. The Image stack keeps track of Image version history, and will always retain at least the original imported Image as the first and original element on the stack.

```
'''
Declare Image img1 found at location filepath with custom dimensions
'''
Image img1 = Image(filepath, 3000, 4000)

'''
Declare Image img2 found at location filepath with original dimensions
'''
Image img2 = Image(filepath)

'''
Declare a blank Image img3 with custom size
'''
Image img3 = Image(3000, 4000)
```

Album

The Album data type is essentially an array of Images. An array of Images displayed in rapid succession can yield an animation use case. Photo# introduces the following syntactic sugar for adding Images to Albums:

```
'''
Declare Album album1 containing Images p1, p2 and add pictures p3, p4
'''
Album album1 = Album(p1, p2)
album1 = album1 + p3
album1 += p4
```

Albums can also be declared empty to start, and have images added to them later.

```
'''
Declare Album album2 containing no Images, then add Image p5 to it
'''
Album album2 = Album()
Album2 += p5
```

Albums can also be added together, combining the Images included in the resulting Album. Images in the right operand will be added after the Images in the left operand in the resulting Album.

```
'''
Add album1 and album2, call the result album3.
album3 will will contain images in the order:
p1, p2, p3, p4, p5
'''
Album album3 = album1 + album2
```

Caption

The Caption data type specifies font, font size, font style, and caption text. Location is specified by a set of keywords {upper, middle, lower} which place the caption in the center of the upper, middle, or lower portion of the image to which it is applied with respect to the center position of the Image.

The upper, middle, and lower keywords may be replaced with an array denoting the start/end location for that argument in the form (x, y), where x is the row and y is the column of the start/end pixel in question.

```
'''
Declare Caption cap, which is located centered on the lower ⅓ of the image
with 0 degrees of counter-clockwise rotation from the x-axis
'''
Caption cap = Caption(“Merry Christmas”, “arial”, 24, “lower”, 0)
```

Gradient

The Gradient data type is a non-destructive color mask that is not a property of an Image, but rather a mask to be linked to an Image.

The upper, middle, and lower keywords may be replaced with a array denoting the start/end location for that argument in the form (x, y), where x is the row and y is the column of the start/end pixel in question.

```
'''
Declare Gradient g with start position “upper” and end position “middle”
'''
Gradient g = Gradient(“upper”, “middle”)
```

IV. Operators and Expressions

Photo# will support all basic mathematical operators involved in manipulating image values. These operators include both arithmetic and logic operators:

Arithmetic Operators

Arithmetic Operators		
Operator	Data Type	Description
+	int, float	plus (addition)
-	int, float	minus (subtraction)
*	int, float	multiply
/	int, float	divide
=	int, float	assignment
%	int	modulo (remainder)

Arithmetic operators consist of +, - , *, / and %.

Photo# supports the += and -= syntactic sugar operators for combining assignment with arithmetic and subtraction, which can be also applied to Photo# composite data types. For integers and floats, the *= and /* syntactic sugar operators are also likewise supported.

Arithmetic operators are left-associative and the highest precedence is the binary *, / and %, followed by the binary + and - operators.

Primitive data types will not have their own set of operators. However, arithmetic operators can only be applied to the same type of primitive data type. For any mismatch of data type, there will be a type error. No modifications will be made to int data types to turn into float data types.

We adopt the same conventions of C in restricting the modulo (%) operator to integers.

Logical Operators

Logical Operators		
Operator	Data Type	Description
&&	boolean	logical and
||	boolean	logical or
!	boolean	logical not
<	int, float	less than
>	int, float	greater than
==	int, float, boolean	is equal to
!=	int, float, boolean	not equal to

Ternary operators and bitwise operators are not supported by Photo#.

V. Image Functions

With the Image data type, Photo# allows for intuitive Image manipulation functions. These functions will implement standard and convenient photo manipulation processes. These processes with all be non-destructive, so the original file will not be overwritten. This is necessary for the integrity of an image should a mistake occur during edit. A copy of the original image before any changes will always be saved. The image edits are saved as separate Image layers on a stack that keeps track of Image version history.

Undo

Pop off an Image from the version history stack, returning the user to the last version of the Image before the most recent edit and save. Takes the top layer off the image stack. Returns the latest edit on the image.

```
# This variable will be used in other examples in this section
Image img1 = Image(filepath)

# Crops the image to an aspect ratio of 16 by 9
Image img1 = crop(img1, 16, 9)

# Undos last edit and returns the image that is removed from the stack

img.undo(img1)
```

Revert

Pop off every Image on the version history stack until the original imported Image is left. Can be specified the amount of reverts to be taken.

```
# reverts the image back one edit
img.revert(img1)

# reverts the image 3 edits back
img.revert(img1, 3)
```

Crop

Images can be cropped by specifying an aspect ratio with which the Image will be cropped with respect to the center pixel. This function takes three arguments: the path of the image, width

and height to crop the image into the specified aspect ratio. The cropped image will be in the aspect ratio given.

```
# How to create the crop edit of the given image.

img.crop(image, width, height)
```

Trim

The trim function takes in parameters for cropping by a specified corner pixel (default to top left pixel) color value, trimming negative spaces based on the color value of the corner pixel. Trims off the pixel and any adjacent pixel of the same color value. For example, trimming a black background on the color value for image will become a blank canvas.

```
'''
How to create the trim edit of the given image.
Takes arguments as the coordinates (x,y) of the pixel and the color value
'''
img.trim(0, 0, black)
```

Caption

Captioning an image applies a Caption data type to an Image. This Caption data type allows the user to apply a description to the Image.

The top, middle, and lower keywords may be replaced with a array denoting the start/end location for that argument in the form (x, y), where x is the row and y is the column of the start/end pixel in question.

```
# How to create the caption edit of the given image.

img.caption(“caption text”, “font name”, font_size, “location”, x-rotate)
```

Duplicate

To create a copy of a given image. The new image will be an exact copy of the given image. The new file will be created in the current directory, but an optional argument of a file path allows the user to create the duplicate in a specific directory.

```
# How to create a duplicate of the given image.
```

```
Image img2 = img.duplicate(img1, filepath)
```

Rotate

Rotation rearranges the Pixels within an Image based on a rotation parameter in units of degrees from the center of the image. The function returns the rotated image.

```
# Rotates the image 100 degrees counterclockwise

img.rotate(100)
```

VI. System Functions

Photo# has built-in system functions as such:

Save

Saves Image file to exported image file after all the edits.

```
'''
If the second argument is not specified (second argument is optional),
overwrite original file.
'''

save(filename, destination_filepath)
```

Import

```
'''
If destination filepath is provided (second argument is optional), the file
will be moved to the new destination before it is imported.
'''

import(origin_filepath, destination_filepath)
```

Print

```
'''
Print can take any data type and print it. The type will be inferred. A
special print output will be provided for Photo# data types
'''

print(args)
```

VII. Control Flow

Related code blocks are indicated based on off-side indentation, similar to Python.

For and While loops

for and while loops are indicated with a beginning for/while statement, followed by the stopping condition. If not iterating through key-worded properties, the syntax is as follows:

```
while boolean:
          Code goes here

for lower_range < i < upper_range:
'''
Provide an enumerator with integer handle 'i' that increments from lower
range to upper range
'''
     Code goes here

for keyword in object:
'''
For loops
'''
     Code goes here

for item in collection:
     Code goes here
```

If and Else statements

Related if/else if/else statements are defined within the same level of indentation. Else statements are attached to the most recent “if”.

```
if condition:
     Code goes here
else if condition:
     Code goes here
else
     Code goes here
```

VIII. User-Defined Functions (Presets)

Photo# supports user-defined functions called *presets*.

```
preset preset_name(arguments):
     Code goes here
```

A sample preset usage shown below:

```
preset summer_vibes(my_album):

     for 0 < i < my_album.length:
     '''
     This will provide enumeration
     from the first to last image in the album
     '''

          for p in my_album.get(i):
               p.brighten(10)
               p.contrast(-5)

               if p == red:
                    saturate(5)
                    hue(-5)
               if p == (0, 100, 100):
                    saturate(10)


          Caption c = Caption(“summer 2019”, “arial”, 20, “lower”)
          '''
          Creates a new caption named c with the indicates parameters
          '''
          my_album.get(i) += c
          '''
          Addition between Photo# data types is supported, and creates
          a composite image. Here, we are layering the caption on top of
          the photo indicated by ‘i’
          '''
```

The above code creates a preset named “summer_vibes” which takes a single photo-album as an argument. It iterates through each image in the album and applies a brightness and contrast edit to each pixel, and a hue and saturation edits to pixels meeting the conditions defined by the if statements. It then creates a new caption and places it on the lower 1/3th (centered) of the image.

In order to call a preset, import the file where the definition is located (if it is in a different file), and call it using filename.presetName(arguments). If the preset is defined in the same file, call it simply with the name of the function in presetName(arguments).

```
'''
Import a helper file helper.p# with the preset preset_name2
defined in it.
'''
import helper.p#

'''
Define a preset preset_name to use within this file
'''
preset preset_name(arguments):
     Code goes here

Image img = Image(“myfile.jpg”)
Image result1 = preset_name(img)
Image result2 = preset_name2(img)
```