# AN INTRODUCTION TO PYTHON

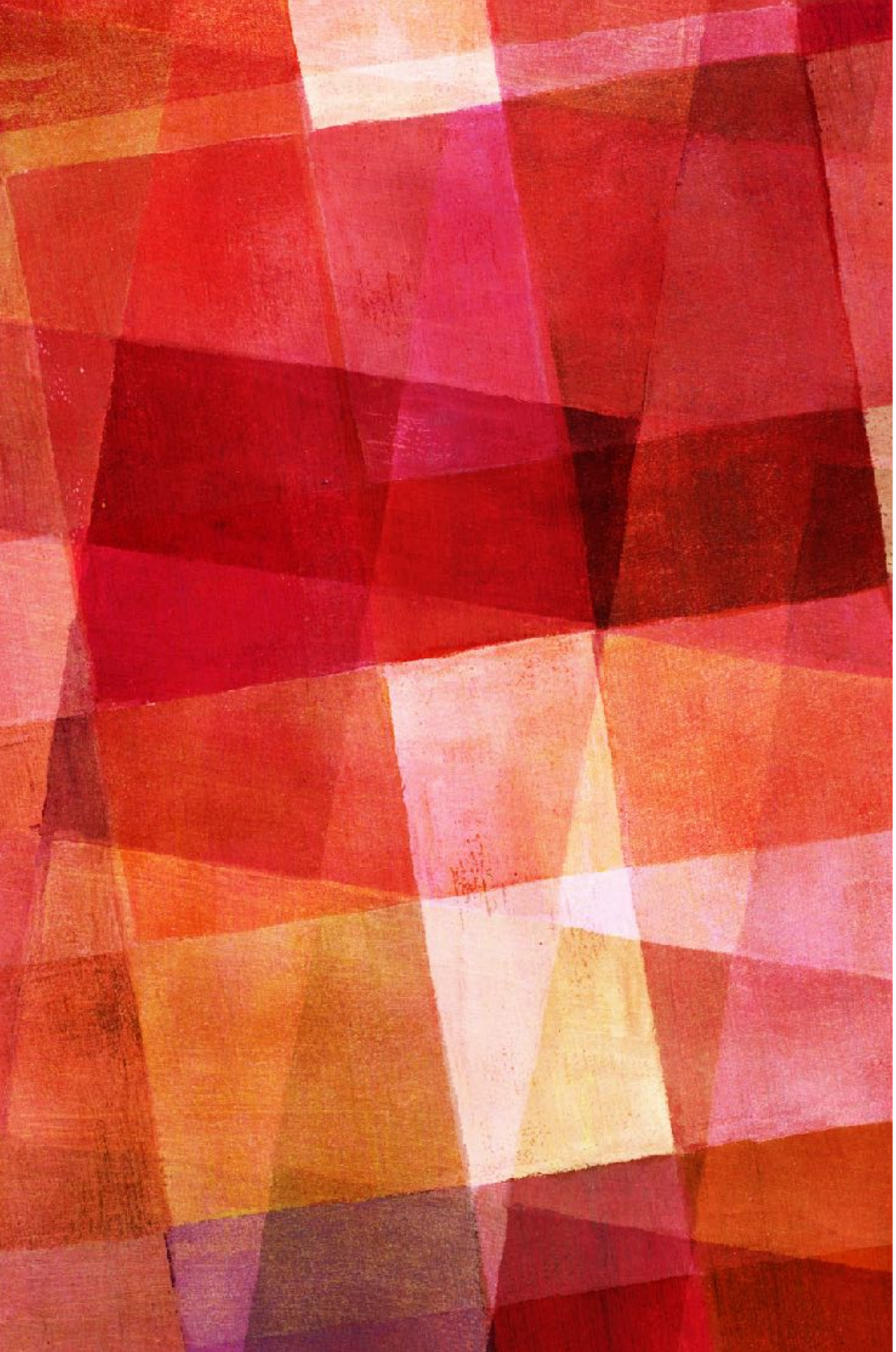
*Ari Maller*

# PYTHON IS A HIGH LEVEL PROGRAMMING AND SCRIPTING LANGUAGE

- Python was designed to be easily understood - straightforward syntax.
- Python is not compiled. It can be run interactively, which is very useful for debugging.
- In spite of this Python need not be slow because routines can be written in C (Cython) and have good performance.
- Pythons true strength comes from the many thousands of packages that can be imported adding high level functionality.
- Python unlike other high level languages (MatLab, Mathematica, Maple, IDL, etc.) is free and runs on all operating systems.

# INSTALLING PYTHON

- The super easy way to install Python is to install Anaconda Python (https://www.continuum.io/downloads) for any OS.
- You can chose to do a full installation which includes 1000+ useful packages or you can do a mini-conda installation which only installs python and conda and then install any packages you want later.
- conda is a package management system that can be used to update python and the packages associated with it
- you can also use pip to install and update packages
- install python 3



# TOOLS

- Terminal
- Text Editors
- IDEs
- Notebooks

# TERMINAL

- Originally a computer only showed text on a terminal. The user could only enter text on one line called the command line.
- While those days are long ago, many programmers still use a terminal to interface with the computer because it tends to be faster and have more control then a graphical interface.
- All OS have a built in program that emulates a terminal.  You can also install other terminal programs with extra features.
- While you do not need to use a terminal ever, understanding how to do things on the command line can be very helpful, especially if you connect to a computer remotely.
- WINDOWS - You must use the Anaconda Terminal for your anaconda python to work.
- MAC OSX- Terminal,  alternative iTerm2

# TEXT EDITOR

- Code is written in text, so to write code you must have a text editor. Vim and Emacs are text editors that work in the terminal and thus can be useful to have used at some point.
- There are many text editors: Notepad++, Emacs, Vim, Atom, Sublime Text, TextWrangler, UltraEdit, Visual Studio Code.
- Syntax Highlighting: This feature means the editor can recognize the type of text you want to write (e.g. python code) and perform color changes and formatting that helps make your code more readable.
- NOTE: Word processors (MS Word, Google Docs) are not text editors. They add a bunch of special characters to have features like bold and italics and these will mess up your code. You can not code in them.

# INTEGRATED DEVELOPMENT ENVIRONMENT

- An IDE combines a text editor and a command line and usually a debugger to create one program with all of your development needs.
- An IDE may only work on one language unlike a text editor which can edit code for any language.
- Some popular ones for Python: Spyder, PyCharm, Rodeo, Atom, Visual Studio (Windows), Xcode (Mac OSX)
- https://wiki.python.org/moin/IntegratedDevelopmentEnvironments
- The distinguish between a IDE and a text editor can be blurred. Many editors can be enhanced with extra packages that make them behave like IDEs

# NOTEBOOKS

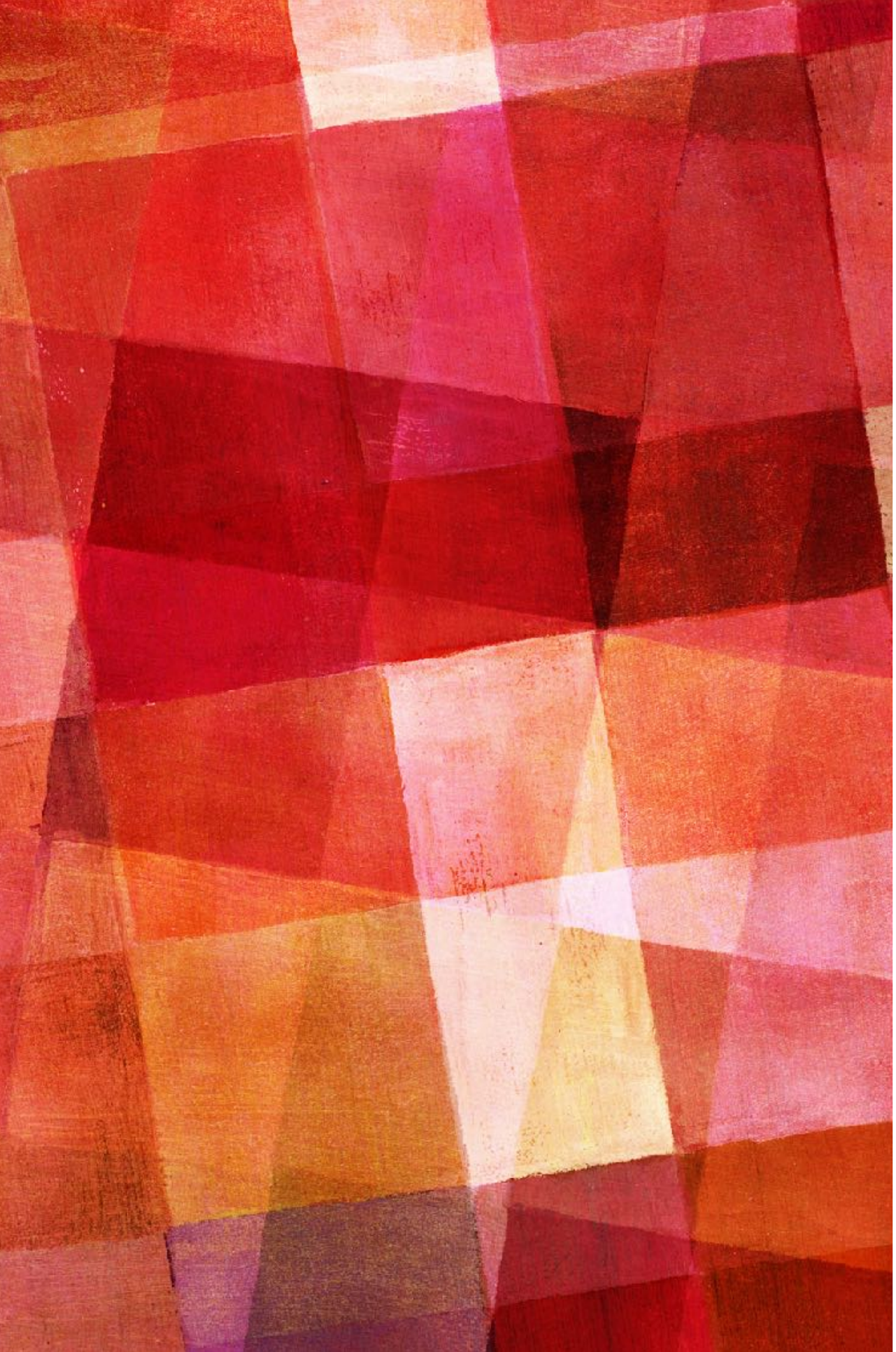
- The final layer of integrating your environment is called a notebook. A notebook not only allows you to write and run code, but also saves all the things that happened when you ran the code.
- Notebooks are very good when first starting a language. Once you know what you are doing you may find them to be overkill.
- The standard notebook for Python is the Jupyter Notebook.
- **conda install jupyter** or **pip install jupyter** or already installed if you downloaded Anaconda Python.
- Jupyter Lab is the newer more featured interface for Notebooks.

# JUPYTER NOTEBOOK

- To start the jupyter notebook enter **jupyter notebook** on the command line or select it from the anaconda navigator.
- This should start a notebook in a web browser. It will also give you a url that you can paste in a browser.
- The notebook will start with a view of your files. You can load an existing notebook or start a new one with buttons on the left side of the page.
- The notebook basically contains cells where you can enter Python code. It also will display images and what you print to standard output.

# JUPYTER NOTEBOOK/LAB

- The notebook has some extra features compared to normal python code, which is nice, but important to distinguish.
- Most importantly you can run any number of lines in a cell and the results are saved into memory. You can then use those results in later cells. Outside of a notebook one generally runs all the lines of code you have at one time.
- The notebook can also capture plots you make, IDEs can also do this, but other ways of running code will open a new window.
- There are some magic commands in the notebook, they start with %. These are notebook features and not in standard python. Using ! in the beginning of a cell is like using a terminal.



# INTRODUCTION TO PYTHON

- variables
- operators
- logic
- control blocks
- sequences
- loops
- dictionaries
- functions
- classes

# DYNAMIC VARIABLES

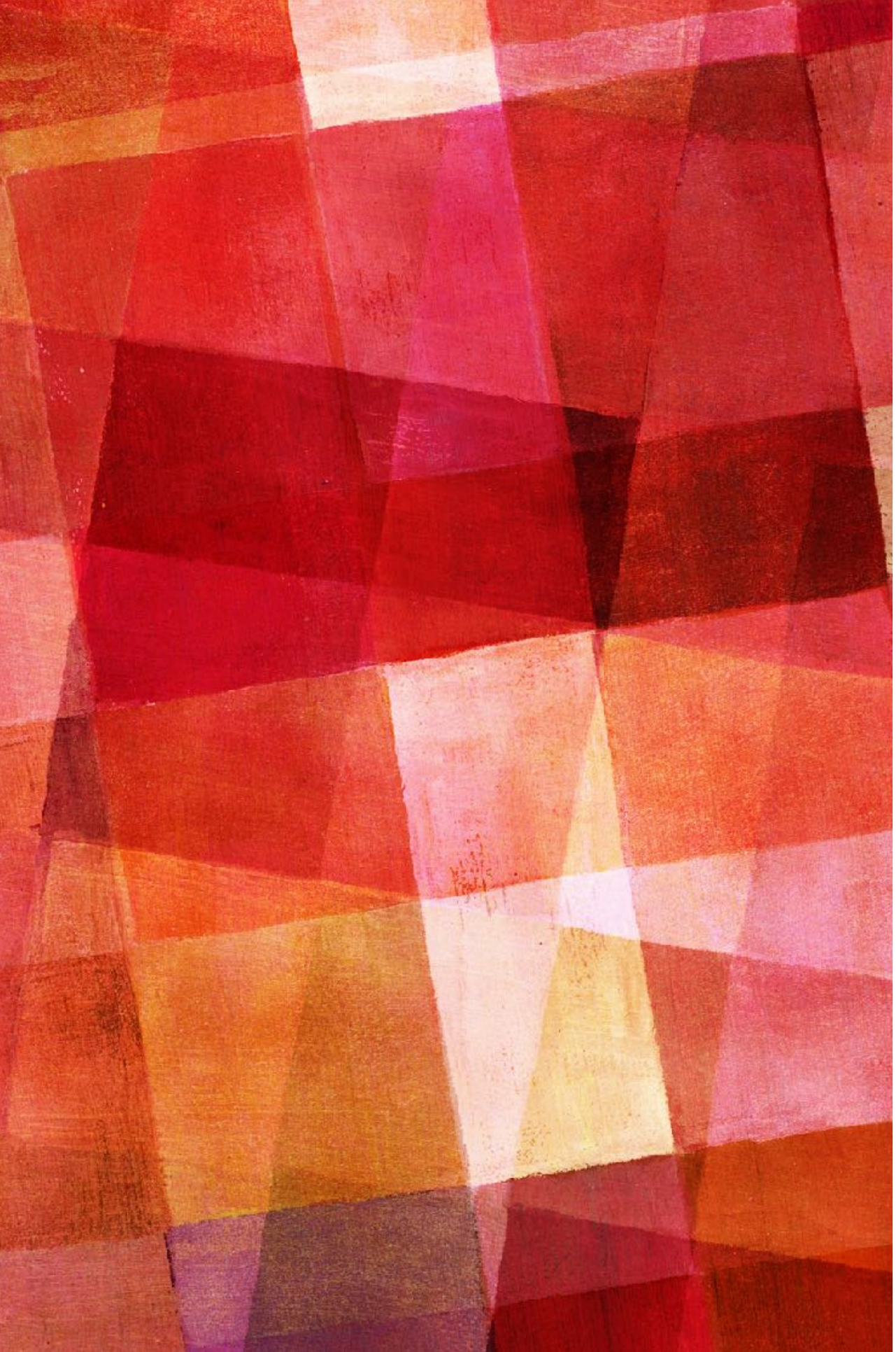
- Variables are not declared, memory is allocated when assigned.
- Types of variables:
  - Boolean: `x=True`
  - Integers: `x=3`
  - Floats: `x=3.1415` or `x=3.e12`
  - Complex: `x=3+5j`
- The type of variable is determined by the value assigned. The same variable's type will change if assigned a different 'type' of value.
- The functions int(), float() and complex() will also set the type. These is called *casting*.
- The function type() will tell you the variable type, very useful for debugging.

# BASIC ARITHMETIC

- x+y  addition
- x-y  subtraction
- x*y  multiplication
- x/y  division
- x**y  exponentiation, raising x to the power y
- x//y  integer division, the number of times y goes into x no fraction
- x%y  modulo, the remainder when x is dived by y
- Parenthesis can be used and are very helpful for complex arithmetic

```
x = (5.0*y+28)**2 + 1
```

- Note you need the * symbol between numbers and variables.



# EXERCISE 1

- Use the quadratic equation to solve $25x^2-35x-15=0$
- Then solve $5x^{2+}3x-90=0$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

# LOGIC

- Python has the following logical comparison operators.

**x==2** returns True if x=2, notice = is assign, == logic test

**x > 2** returns True if x greater than 2

**x >=2** returns True if x greater than or equal to 2

**x< 2** returns True if x less than 2

**x <=2** returns True if x less than or equal to2

**x!=2** returns True if x not equal to 2

- Python also has the operators; and, or and not to create syntax like

**x > 2 and x < 4** or **x==8 or not x > 5**

- In addition Python has **is** which is slightly different than ==.

**x is None**

- The difference between is and == is that == evaluates if two things have the same value, is checks if it is the same address in memory. So

**x=4**

**x is 4**

will return False. You should use is for None, True and False checks only.

**if x:**

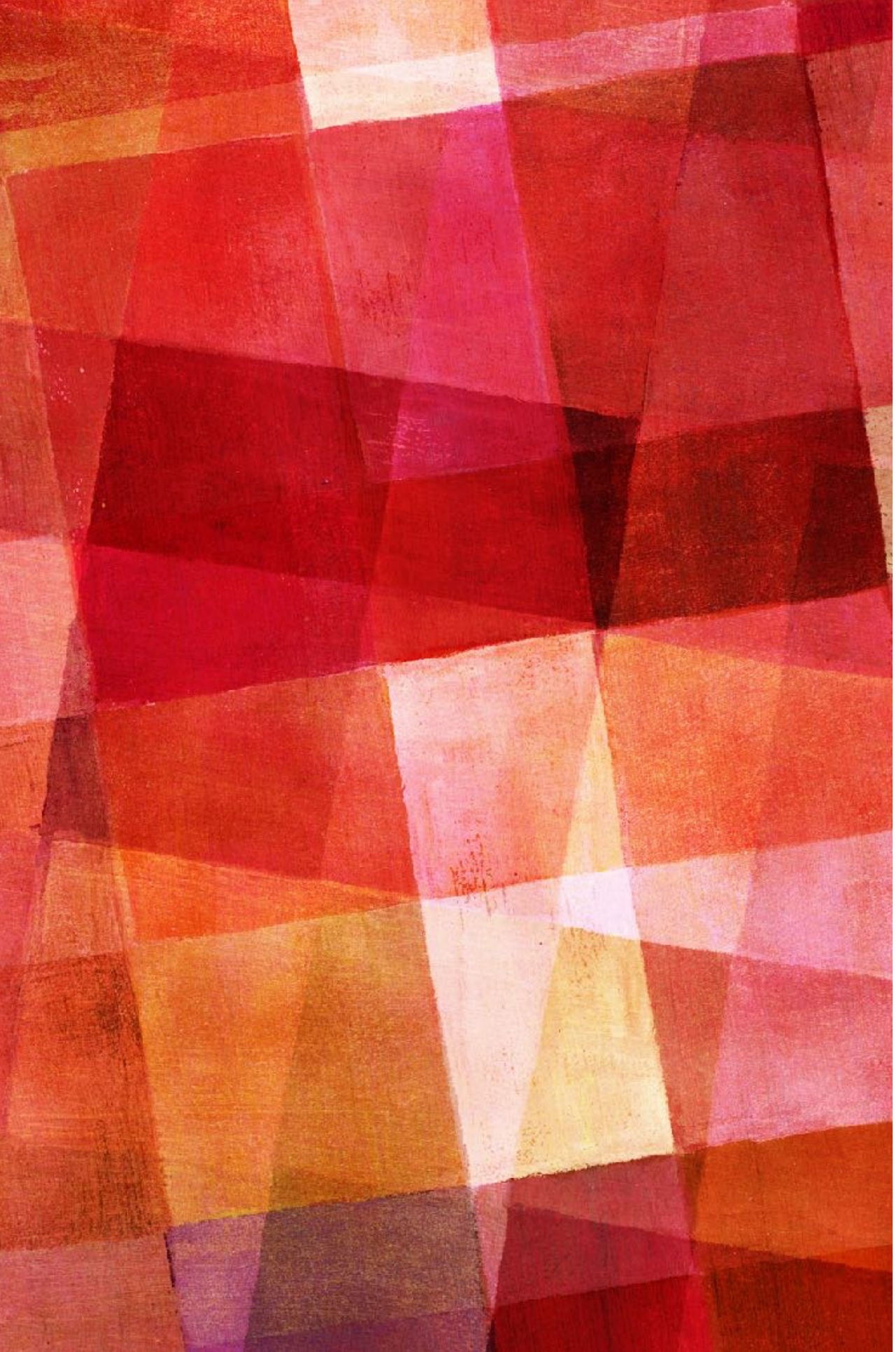
checks if x is set to a nonzero value, this can be unclear if not set to True or False.

# CONTROL BLOCKS

- In Python control blocks are created by a statement ending with a colon and then all text that is intended is in that block.

```
if (x==4):
    print("x is equal to four")
elif(x==5):   #else if
    print("x is equal to five")
else:
    print("x is not equal to four or five")
```

- Comments are made with the # symbol. Any text after a # is ignored.
- Long comments can also be made with the triple quote ''' though this supposed to be for docstrings.



# EXERCISE 2

- Write a program to test if a variable is positive, negative or zero.

# WHILE

- Instead of checking a condition just once we can keep checking using a while loop.

```
x=0
while(x<5):
    print(x)
    x=x+1
print("Got to x=4")
```

or

```
while True:
    print (x)
    x=x+1
    if (x==4):
        break
print("Got to x=4")
```

# SEQUENCES (STRINGS AND LISTS)

- Often we may have a whole bunch of numbers or things we would like to do something with. The most obvious example of this are words, a sequence of characters. These use the string type in Python.
- A more general type of sequence is a list, which does not need to be characters but can be anything stored in memory. Lists are created with square brackets, [].
- You can get the elements in a sequence using indices and the square brackets.

# STRINGS

- Strings are an example of a sequence and are set using quotation marks.
- **x='Hello'** or **x="Goodbye"**
- **x="1.0"** is also a string not a float. **x+1.0** will give an error since it is not clear what you want to do. Same as if you tried **"Hello"+1.0**.
- You can convert a string to a variable with the int(),float() or complex() commands.
- You can access the elements in a string; if **x='Hello'** then **x[0]='H'** and **x[2]='l' and x[1:4] = 'ell'**

# LISTS

- Lists can be sequences of anything, including other lists.

```
shopping = [‘eggs','beer','milk']

numbers = [1,2,3,4,5]

stuff = [1,2.e-2,'four',[0,1]] #a list can contain another list
```

- Sequences can be sliced, that means have certain elements returned

```
odd=numbers[0:5:2] #[start,stop,step] step defaults to 1

drinks = shopping[1:] #start defaults to 0, stop to last value

shopping[1] = ‘wine’ #lists can be changed

shopping = shopping + [‘cheese’] #and added to
```

# FOR LOOP

- One of the main uses of sequences is that they can be iterated over. In Python iteration is favored over indexing. So a for loop might look like

```
list=[1,2,3,4,5]

for num in list:
    print(np.sqrt(num))


names=[“Bob”,”Sally”,”Vicky”]
for name in names:
    print(“Hello ”+name)
```

# FOR LOOP TRICKS

- create list of integers to loop over

```
for i in range(len(array)): #len() returns length of a list or array
```

- add integers to a list and loop over both

```
for i,name in enumerate(names):
```

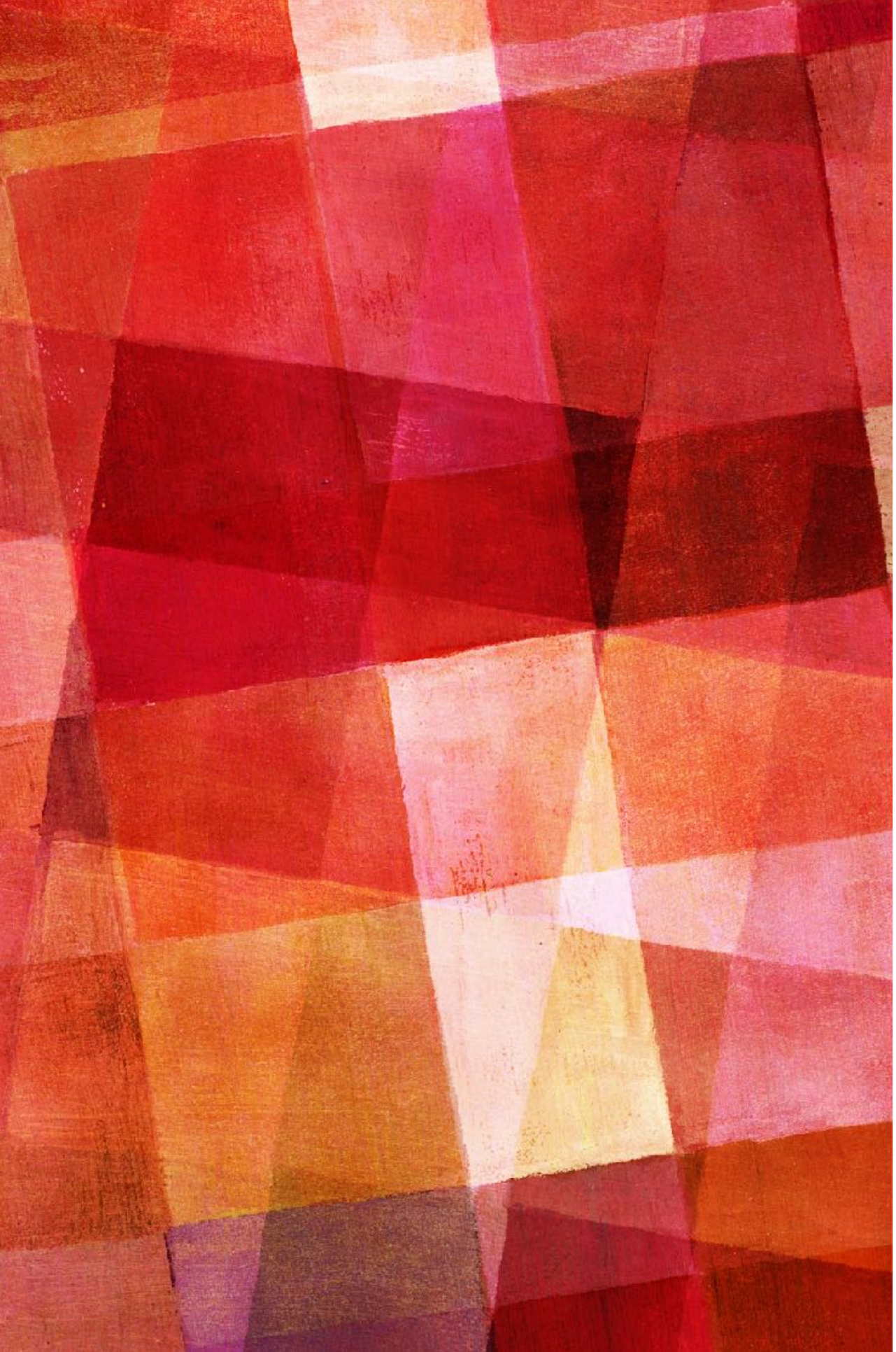
- combine two lists and iterate over both

```
for thing,name in zip(things,names):
```

- single line for loops (called a list comprehension)

```
doubled = [num*2 for num in array]
```

```
evendouble=[num*2 for num in array if num%2=0]
```



# EXERCISE 3

- Write a loop that prints the odd numbers up to 20

# DICTIONARIES

- A dictionary is a pairing between values and keys. You give the dictionary the key and it returns the value. This data type is not often used in beginning programming, but it can be very useful.

```
astro_type={'star':1,'planet':2,'galaxy':3,'black hole':4}
print(astro_type['star'])
print(astro_type['black hole’])
```

# DICTIONARIES

- One good use of dictionaries is to replace long if/else blocks.

```
if object==‘star’:
    stars=stars+1
elif object==‘galaxy’:
    gals=gals+1
elif object==‘planet’:
    planets=planets+1
else:
    unknown=unknown+1
```

- this can be replaced with a dictionary and a list

```
obj_type=[0,0,0,0] #star,galaxy,planet,unkown
obj_dict={‘star’:0,’galaxy:1,’planet’:2,’unknown’:3}
obj_type[obj_dict[object]]=obj_type[obj_dict[object]]+1
```

# FUNCTIONS

- A function is some code that can operate on some inputs and can return some outputs. We have already used many functions; print(), range(), int(), type(), enumerate(). You'll notice all functions have parenthesis at the end in which you can pass variables to be used by the function. We will now define our own functions.

- To define your own functions in Python you use the def syntax.

```
def factorial(n):
    f=1.0
    for k in range(1,n+1):
        f*=k     #  *=,+=, are shorthand for f=f*k, f=f+k
    return f
```

- Like in a control block, indention determines when your function ends.
- Functions must be defined before they are used.

# ARGUMENTS

- Functions have two main types of arguments, positional arguments and keyword arguments. As the names imply, positional arguments are set by position, while keyword arguments are set by keyword. Also, you must have the same number of positional arguments as defined in the function, but keyword arguments can be omitted.
- To pass information to your function you can use positional arguments,

**def factorial(n):**

- this passes one argument that will be called n in your function. If you don't pass a value this will give an error. You can have any number of arguments passed including 0

**def nada():**

**def cool_function(n1,n2,n3):**

- These arguments are called positional because they are assigned in the order you pass them.

**cool_function(3,5,7):**

- will assign n1=3, n2=5 and n3=7. Note it doesn't matter at all what the names of the variables are, only their order.

# KEYWORD ARGUMENTS

- you can also have keyword arguments that don't need to be, but can be passed

```
def sphere2cartesian(R, theta, phi, degrees=False)
    if degrees:
        deg2rad=np.pi/180.
        R=deg2rad*R; theta=deg2rad*theta; phi=deg2rad*phi
    x=R*np.cos(theta)*np.sin(phi)
    y=R*np.sin(theta)*np.sin(phi)
    z=R*np.cos(phi)
return x,y,z
x,y,z=sphere2cartesian(100., 45., 30., degrees=True)
```

# UNSPECIFIED ARGUMENTS

- You can also pass unspecified variables and keyword arguments to a function. This is done using the *args and *kwargs terms. These are less clear and should only be used when necessary.

```
def myfunc(*args,**kwargs):
    for arg in args:
        print(arg)
plt.plot(x,y,**kwargs)
```

# RETURN

- Your function can return information to the code that calls it using the return statement

```
def seven():
    a=7
  return a
```

- You can return any number of values of any data type or nothing at all.

```
return a, b, c, d
```

- When your code hits return it exits the function so it will not execute any lines below the return statement

```
def seven(n):
    return 7
    print(“this line will never be printed”)
```

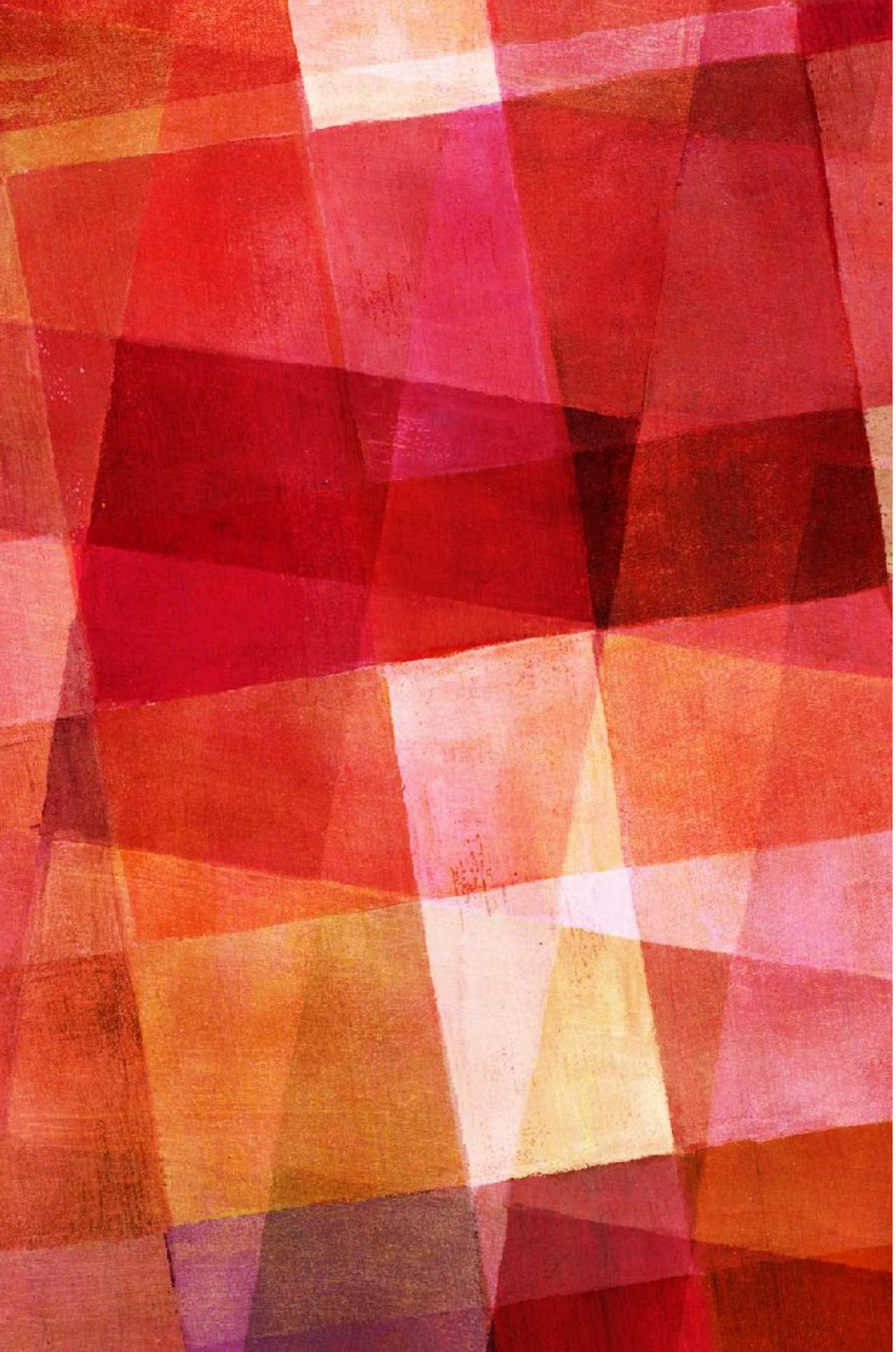
- If you want your function to return values to a variable you need to use the =

```
v=seven()
```

- assigns the value of 7 to the variable v. For multiple variables assign to a list or multiple vars

```
a,b,c,d = my_function(7,8,9,10)
list = my_function(7,8,9,10)
```

- an error will be raised if the number of things returned does not equal the number of assignments.



# EXERCISE 4

- Write a function that given the height of a ball determines the time it takes to hit the ground. $h = 1/2\ g\ t^2$. Allow g to be a keyword so you can use this code on other planets.

# CLASSES

- Everything in python is actually a class. But you shouldn't worry too much about classes or object oriented programming when you are just getting started. However, understanding a bit about classes can help you understand how to use python. The main idea of object oriented programming is that data and functions should be combined together. A function that is in a class is called a method. Since everything in python is a class, each data type we have seen is a class. We can see the methods of the class with the dir() function.

```
x = 6.5

dir(x)
```

- methods with the __name__ format are internal and not meant to be used. So we see for floats we have methods like as_integer_ratio() and is_integer(). We can use these methods on our float

```
print(x.as_integer_ratio())

print(x.is_integer())
```

# METHODS FOR LISTS

- Most of the methods on variables are not that interesting, but for list and strings they can be very useful. For lists we have methods like, append(), insert(), pop() and remove() which allow us to modify our list.

```
x=[] #create an empty list
x.append(1) #append adds a value to your list
x.append(2)
x.append(1)
x.insert(1,'new') #insert places a value into the list
y = x.pop(2) # pop gets the value and removes it from the list
x.remove(3) #remove just removes it
```

# METHODS FOR STRINGS

- Strings also have a bunch of useful methods attached to them like find(), rfind() and split().

```
x='Hello World’

a=x.find('W')

b=x.rfind('d')+1

print(x[a:b]) # prints World

print(x.split()) # returns [‘Hello’,’World’]

print(x.swapcase()) # returns ‘hELLO wORLD’
```

# WRITING CLASSES

- Writing a class isn't all that complicated. One just creates a class and then defines your methods inside of that class. There is the special name self that refers to an instance of the class and some special methods like __init__which runs when you create an instance of the class. In general, 2 underscores before and after a name in python are used for special types of function of variable name being used internally. Here is an example of how to code a class.

```
class Student:
    def __init__(self, name, grade, major): #__init__ is a special name
        #that will be run when creating an instance of the class
        self.name = name
        self.grade = grade  #self is the instance of the class
        self.major = major

    def passing(self):
        if self.grade=='A' or self.grade=='B' or self.grade=='C':
            return True
        elif self.grade=='D' and self.major=='English':
            return True
        else:
            return False


student1=Student('Jill','A','English') #this is an instance of the class
student2=Student('Beth','F','Biology') #this is another one
print(student1.name,student1.grade)
print(student1.passing(),student2.passing())
```

# PACKAGES AND MODULES

- Python has relatively few built in functions and data types. Its functionality is greatly increased by a packages developed by individuals. There are a few ways to bring functions and classes from a package into your program

**from numpy import exp,log** adds the functions exp() and log()

**from numpy import *** adds all functions from numpy

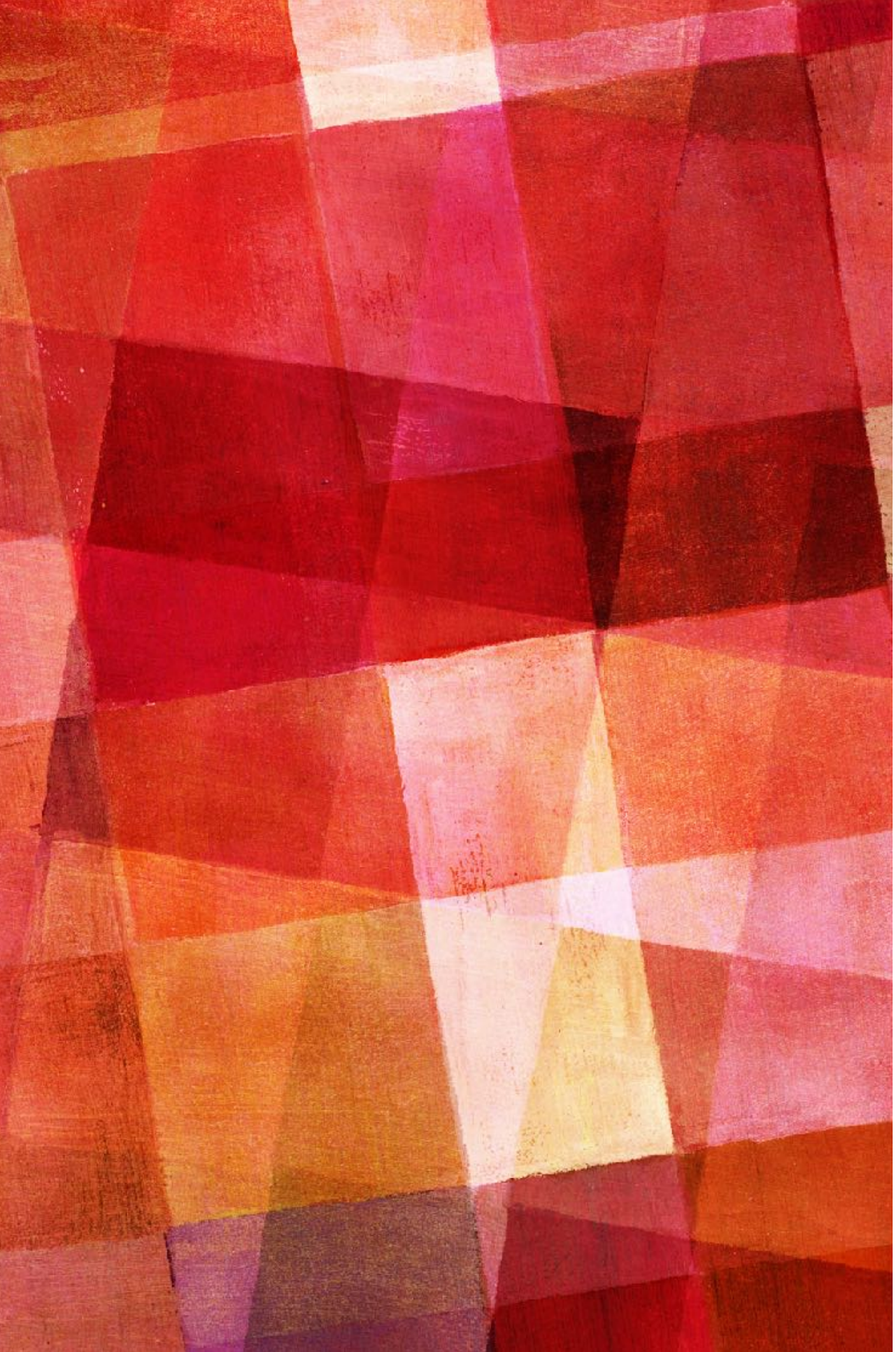
**import numpy** adds all functions, callable as numpy.exp()

**import numpy as np** adds all functions, callable as np.exp()

- I strongly discourage the first two because they make it less clear where the functions came from. Also you can import two functions with the same name. Note you can also import a submodule like,

**import scipy.optimize as opt**

- The built in functions dir() and help() can be very useful with modules. dir() with no arguments returns all currently available functions, dir(module or variable) returns all functions or methods associated with that module or variable. help() returns helpful information (as written by the coder) about the module or variable.



# PACKAGE MANAGEMENT

- The thousands of packages available in Python are the languages greatest strength.
- However, it can be very hard to keep track of them and people constantly update them so they don't stay constant.
- A package management system is code that helps you with this, conda and pip are package management systems.

# CONDA

- Conda is a way to add and manage the packages in your python distribution. Some useful commands

**conda list [name]**

- lists the package name if installed [or all packages installed]

**conda search name**

- searches for packages with this name

**conda install name**

- installs the package name

**conda update name**

- updates the package name

# CHANNELS

- The anaconda team maintains the default channel for your conda installation. However, you may find that you want some package that is not included in the default installation.
- There are other channels that you can use to find packages, the most common alternative to default is conda-forge. This channel tries to get updates out faster and new packages into the channel quicker.
- To search or install a package from a different channel use -c conda-forge after the command.
- If you only install a few packages from a different channel you will probably be all right, but the more you do the more likely this will cause problems. If you really want to use conda-forge you set it as your default channel. If you do that also set your channel_priority to strict.

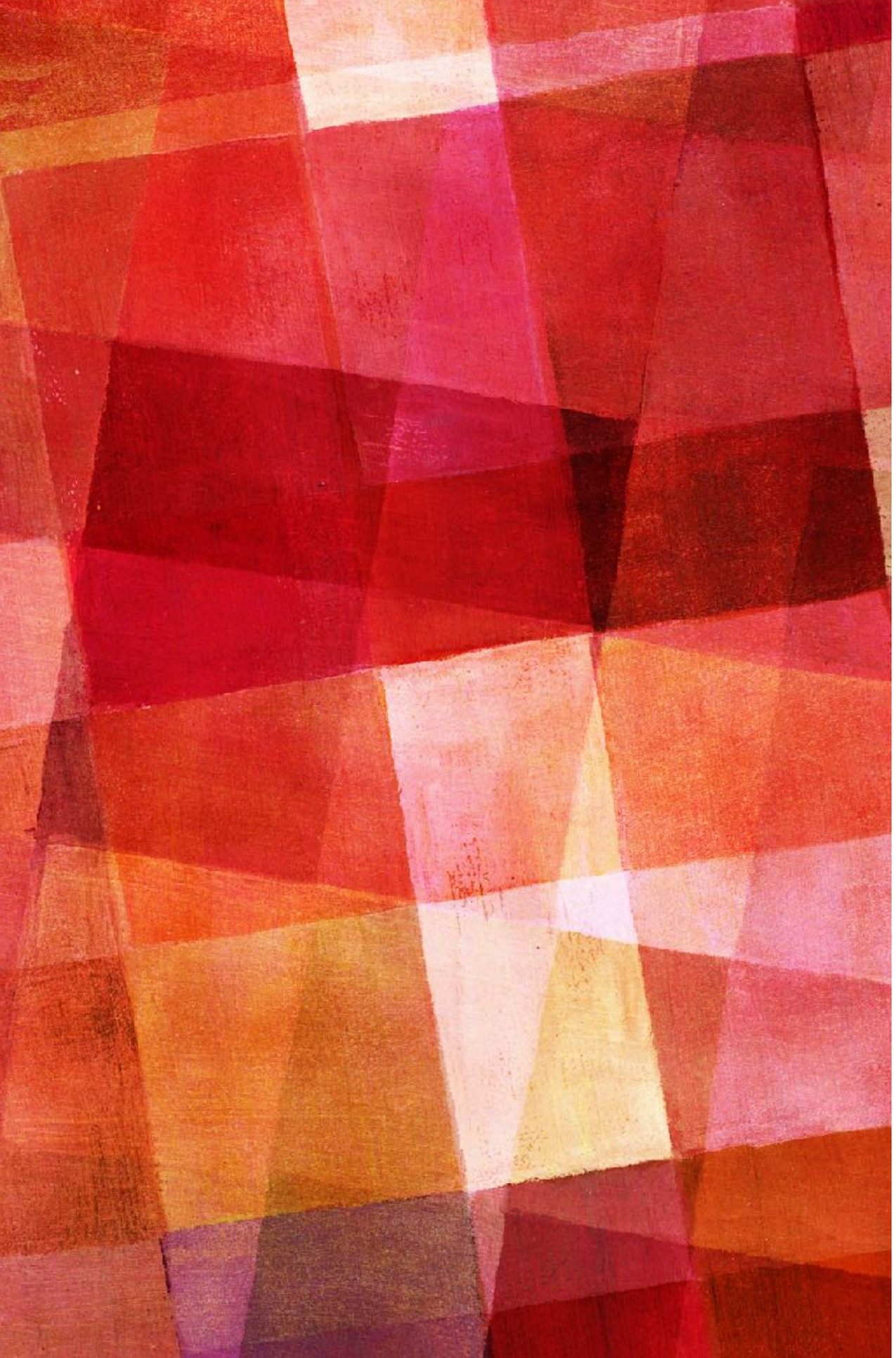
conda config --add channels conda-forge

conda config --set channel_priority strict

# PIP

- An alternative package manager is pip. Pip is only for python and has a somewhat larger selection of packages. However, pip doesn't know about conda while conda knows about pip. So I tend to try conda first and then use pip if conda doesn't work.
- Pip has similar commands but slightly different

pip freeze
pip search name
pip install name
pip install —upgrade name



# PYTHON SCRIPTS

- Often it is useful to be able to run your code from the command line.
- This is why python is called a scripting language, because one can run lines of code without having to open an IDE or notebook.
- Any code that you run from the command line could already be a python script.
- Our goal here is to have our code run if we type **python mycode.py**

# PYTHON SCRIPTS

- There are two ways to run Python as a script. One is to run Python and then pass the script as an argument

**python my_script.py**

- The second is to just type the script file's name if it is in your PATH and is executable.

**./my_script.py**

- The .py extension is not necessary, both of the above cases can work without it. The extension lets the OS know what type of file you have and the OS can do certain things like invoke the Python interpreter or use syntax highlighting based on that. Also it lets others know your file is Python code.

- You should also include the following as the first line in your code

**!# /usr/bin/env python** or **!# /usr/bin/python**

- this also tells the shell the file is Python. If you use the .py extension it is not really needed, though you can use it to specify what version of Python. It is mostly historic, but also tells a reader that the file will be Python code and is being thought of as a script.

# EXECUTABLE

- To make a file executable in Linux or OS X do

**chmod +x myfile**

- In Windows the .py extension will tell the OS that you want the file to be executable.
- With Linux/OS X if you want a whole directory of files to be easily executable you can add the directory to your path. How to do this depends on the shell you are using, which will usually be Bash. Changing your path will be something like

**export PATH="/Users/me/bin:$PATH"**

- make sure you include everything that was already in your old PATH, otherwise all commands you are used to running will no longer work.

# IS __MAIN__

- For Python scripts it is nice to add a check that you are actually trying to run this code as a script and not for example in a notebook.
- This can be done with the following line of code

```
if __name__=="__main__":
    main()
```

- This checks that the code is being run from the command line, in which case __name__ will be set to __main__. So if that is true if will execute what follows, but if not then it won't.
- The main point of this is if you have some functions you define in your file, you might want to import them into some other file, or into a interactive session, but not run the program. This check allows that to happen.

# SYS.ARGV

- Next step, we want to run our program, but we also want to be able to change some options.
- One way to do this is with the input function that allows the code to ask for an input.
- Another way to do this is to pass options on the command line like other command line programs.
- Anything you type in the command line after your file name is saved as a string which can be accessed with the sys package as sys.argv.
- However a package exists to help you parse that string which can be much easier to use if you have complicated options.

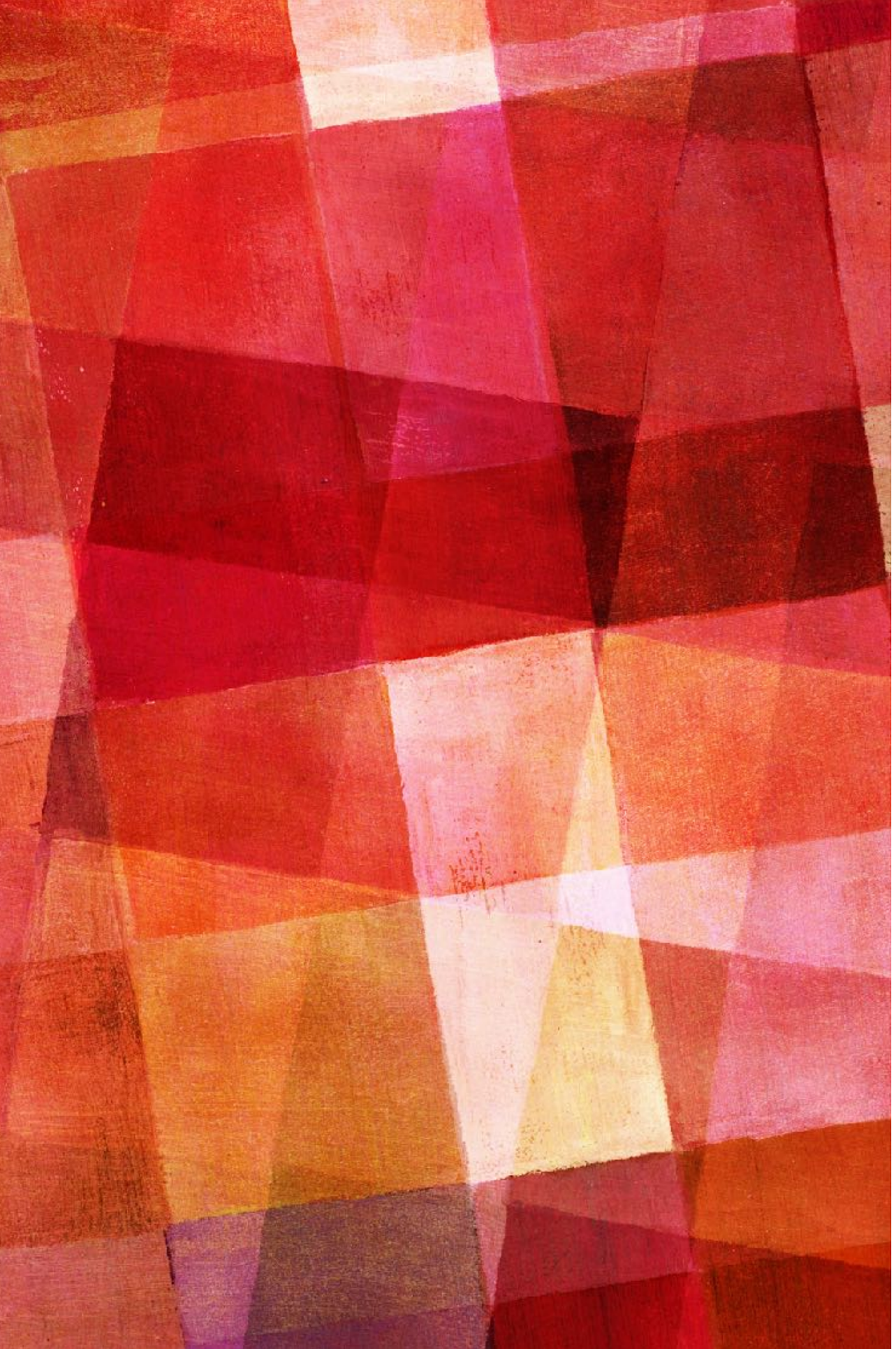
# ARGPARSE

- The argparse package works like this:
  - First you have to create a parser
  - Then you add arguments to it
  - Finally you call the parse_args() function and it returns your arguments.
- One of the key reasons to use argparse is that you should give a line of explanation for each argument that will be printed to the screen if there is something wrong with what you type on the command line.
- Argparse has a lot of options, we will start with just the basics.

# ARGPARSE

```
import argparse

parser = argparse.ArgumentParser(description="Do Something.”)

parser.add_argument(‘h’, type=float, help=‘height of object above ground’)

parser.add_argument(’N’,type=int, help=‘Number of bins for integrating’)

parser.add_argument(“—simp”,default=False, help='Use Simpson’s Rule’,
        action= "store_true")

args=parser.parse_args()

print(args.h, args.N, args.simp)
```

- If the name of this file was test.py I could then type python test.py 30.0 100 and then it would print, 30.0 100 False



# VERSION CONTROL

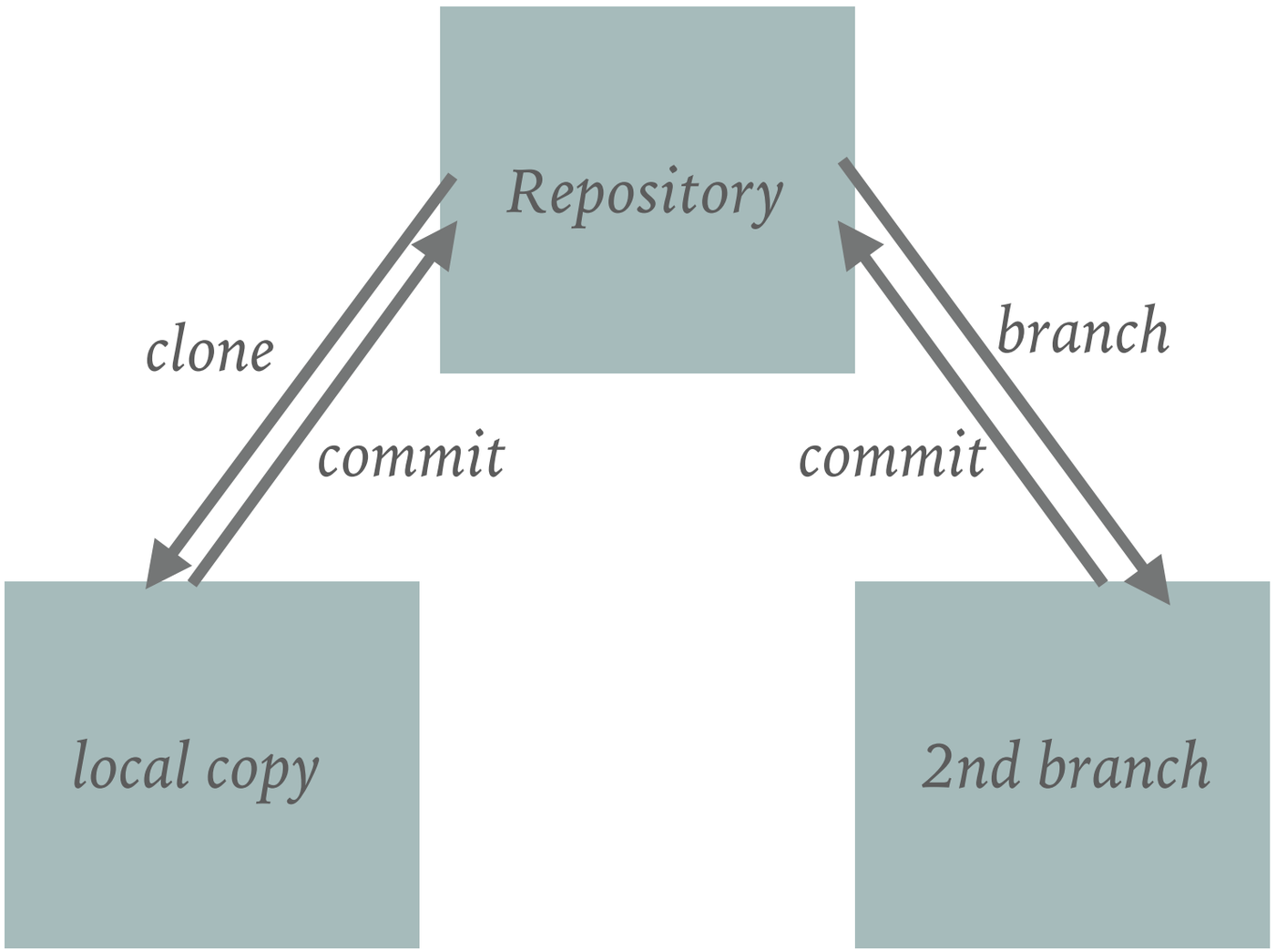
- A way to keep track of changes to your code.
- Also a way to collaborate on writing code with others.
- Many implementations though git is becoming the most common one.
- Also can be combined with a website to allow easy access to others, even people you don't know.
- Widely used in open source programming.

# VERSION CONTROL

- Someday you will spend a long amount of time writing a fairly large and complicated code.  When it runs and all the bugs have been worked out you will be very happy.
- Then you will want to make some minor revisions to your code, but after you do your code will no longer work.
- Not being able to figure out exactly what changes you made to the formally working version you will get very frustrated and wish you had saved a copy of the version that worked.
- THIS IS WHY WE USE VERSION CONTROL!

# VERSION CONTROL

- The basic idea of version control is that your files are kept in what is called a repository.
- However, you do not edit the files in the repository. Instead you check out a copy of them for you to work on.
- When you are done working on the files you commit them back to the repository.
- The version control software keeps track of each of these changes and allows you to check out an older copy if you want.
- It is also possible to create branches, where you change different things, or different people change different things and then merge them back at some later time.


Repository
clone
commit
branch
commit
local copy
2nd branch

# GIT

- The most popular version control system is probably git, and that has a lot to do with GitHub a website that freely hosts your files as long as they are open source.
- git can be installed from a package management system (like conda) or with a GUI. Here is one possible choice: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
- You can either use the command line or a GUI, they do the same things. You should at least practice on the command line so you understand what the GUI is doing.
- Installing Anaconda Python should have installed git.

# GIT WORKFLOW

- git init  - create an new repository / or create a repository on GitHub using the web interface.
- git clone <repo url> - make a local copy of the repository
- git add <filename> - adds file or updates file to local copy
- git commit - updates changes to the repository
- git push - uploads changes to remote repository

- Then when you want to checkout the updated version of the repo you just need to use
- git pull

# Git Cheat Sheet

## Git Basics

| | |
|---|---|
| `git init <directory>` | Create empty Git repo in specified directory. Run with no arguments to initialize the current directory as a git repository. |
| `git clone <repo>` | Clone repo located at `<repo>` onto local machine. Original repo can be located on the local filesystem or on a remote machine via `HTTP` or `SSH`. |
| `git config user.name <name>` | Define author name to be used for all commits in current repo. Devs commonly use `--global` flag to set config options for current user. |
| `git add <directory>` | Stage all changes in `<directory>` for the next commit. Replace `<directory>` with a `<file>` to change a specific file. |
| `git commit -m "<message>"` | Commit the staged snapshot, but instead of launching a text editor, use `<message>` as the commit message. |
| `git status` | List which files are staged, unstaged, and untracked. |
| `git log` | Display the entire commit history using the default format. For customization see additional options. |
| `git diff` | Show unstaged changes between your index and working directory. |

## Undoing Changes

| | |
|---|---|
| `git revert <commit>` | Create new commit that undoes all of the changes made in `<commit>`, then apply it to the current branch. |
| `git reset <file>` | Remove `<file>` from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes. |
| `git clean -n` | Shows which files would be removed from working directory. Use the `-f` flag in place of the `-n` flag to execute the clean. |

## Rewriting Git History

| | |
|---|---|
| `git commit --amend` | Replace the last commit with the staged changes and last commit combined. Use with nothing staged to edit the last commit's message. |
| `git rebase <base>` | Rebase the current branch onto `<base>`. `<base>` can be a commit ID, a branch name, a tag, or a relative reference to `HEAD`. |
| `git reflog` | Show a log of changes to the local repository's `HEAD`. Add `--relative-date` flag to show date info or `--all` to show all refs. |

## Git Branches

| | |
|---|---|
| `git branch` | List all of the branches in your repo. Add a `<branch>` argument to create a new branch with the name `<branch>`. |
| `git checkout -b <branch>` | Create and check out a new branch named `<branch>`. Drop the `-b` flag to checkout an existing branch. |
| `git merge <branch>` | Merge `<branch>` into the current branch. |

## Remote Repositories

| | |
|---|---|
| `git remote add <name> <url>` | Create a new connection to a remote repo. After adding a remote, you can use `<name>` as a shortcut for `<url>` in other commands. |
| `git fetch <remote> <branch>` | Fetches a specific `<branch>`, from the repo. Leave off `<branch>` to fetch all remote refs. |
| `git pull <remote>` | Fetch the specified remote's copy of current branch and immediately merge it into the local copy. |
| `git push <remote> <branch>` | Push the branch to `<remote>`, along with necessary commits and objects. Creates named branch in the remote repo if it doesn't exist. |

Atlassian

**Visit atlassian.com/git for more information, training, and tutorials**