# Material Summary: Basic Algebra

## 1. Polynomials

* We already looked at **linear** and **quadratic polynomials**
* **Term (monomial):**
  + Coefficient (number), variable, power (number )
* **Polynomial**: sum of monomials
  + Degree: the highest degree of the variable (with coefficient )
* **Operations**
  + Defined the same way as with numbers
  + **Addition** and **subtraction**
  + **Multiplication** and **division**

## 2. Polynomials in Python

* **numpy** has a module for working with polynomials
  + Includes the "general" polynomials, as well as a few special cases
    - Chebyshev, Legandre, Hermit
* **Storing polynomials**
  + As arrays (index power, value coefficient)
  + Keep in mind this will look "reversed" relative to the way we write

import numpy.polynomial.polynomial as p

p.polyadd([-8, 5, 2], [-2, 0, 0, 0, 3])

p.polymul([-8, 5, 2], [-2, 0, 0, 0, 3])

# array([-10., 5., 2., 0., 3.])

# array([ 16., -10., -4., 0., -24., 15., 6.])

* **Pretty printing**
  + Use **sympy** to print the polynomial
    - If it's a list, use it **directly**
    - If it's a Polynomial object, call the **coef property**
  + Reverse the order of coefficients (sympy expects them from highest to lowest)

import sympy

from sympy.abc import x

polynomial = p.Polynomial([-2, 0, 0, 0, 3])

sympy.init\_printing()

print(sympy.Poly(reversed(polynomial.coef), x).as\_expr())

# Output: 3.0\*x\*\*4 - 2.0

## 3. Set

* An **unordered collection** of things
  + Usually, numbers
  + No repetitions
* **Set notation**:
  + "**The set of numbers x, which are a subset of the real numbers, which are greater than or equal to zero**"
  + **Left**: example element
  + **Right**: conditions to satisfy
* Python **set comprehensions**
  + Very similar to what we already wrote
  + Also very similar to list comprehensions (but with curly braces)

positive\_x = {x for x in range(-5, 5) if x >= 0}

# {0, 1, 2, 3, 4}

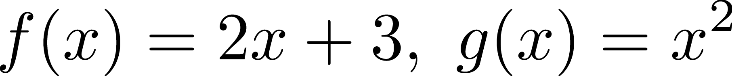
* **Cardinality**: number of elements
* Checking whether an **element is in the set**: 
* Checking whether a **set is subset of another set**: 
* **Union**: 
* **Intersection**: 
* **Difference**: 

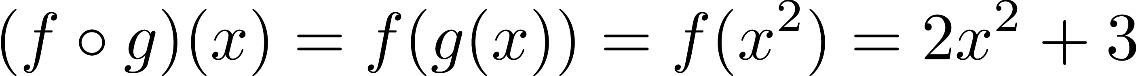
## 4. Functions

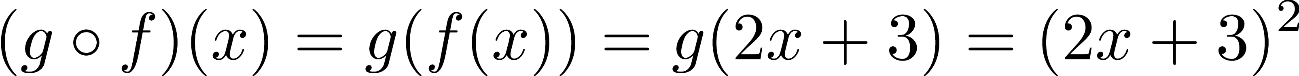
* A relation between set of inputs *X* (**domain**) and a set of outputs *Y* (**codomain**)
* **One input produces exactly one output**
* The inputs don't need to be numbers
* Functions don't know how to compute the output, they're just mappings
* In programming, we write **procedures**
* Math notation:
  + Commonly abbreviated as: 
* Some more definitions:
  + **Injective** (one-to-one): unique inputs => unique outputs
  + **Surjective** (onto): every element in the codomain is mapped
  + **Bijective** (one-to-one correspondence): injective and surjective
  + Here is [a graphical view](https://www.mathsisfun.com/sets/injective-surjective-bijective.html)

## 5. Function Composition

* Also called **pipelining** in most languages
* Takes two functions and applies them in order
  + **Innermost to outermost**
  + **Math notation**: 
  + Can be generalized to more functions
* Note that the order matters







* This kind of notation can be confusing sometimes
  + is only a placeholder for the input
  + We've used the same letter for different inputs
  + **Tip**: When working with complicated functions, be very careful what the inputs and outputs are, and how variables depend on other variables
* Functions and composition are the basis of [functional programming](https://en.wikipedia.org/wiki/Functional_programming)

## 6. Function Graphs (Plots)

* One very intuitive way to get to **know functions is to plot them**
  + Generate values in the **domain** (independent variable)
  + For each value **compute the output** (dependent variable)
  + Create a **graph**
  + Plot all computed points and connect them with tiny straight lines
* **lambda** in Python is a short syntax for a function
  + We can define it outside as well (it's just shorter and simpler to use it inline)

import numpy as np

import matplotlib.pyplot as plt

def plot\_function(f, x\_min = -10, x\_max = 10, n\_values = 2000):

x = np.linspace(x\_min, x\_max, n\_values)

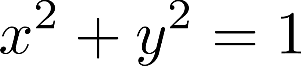
y = f(x)

plt.plot(x, y)

plt.show()

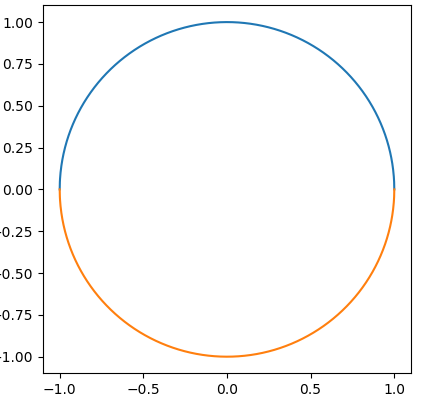
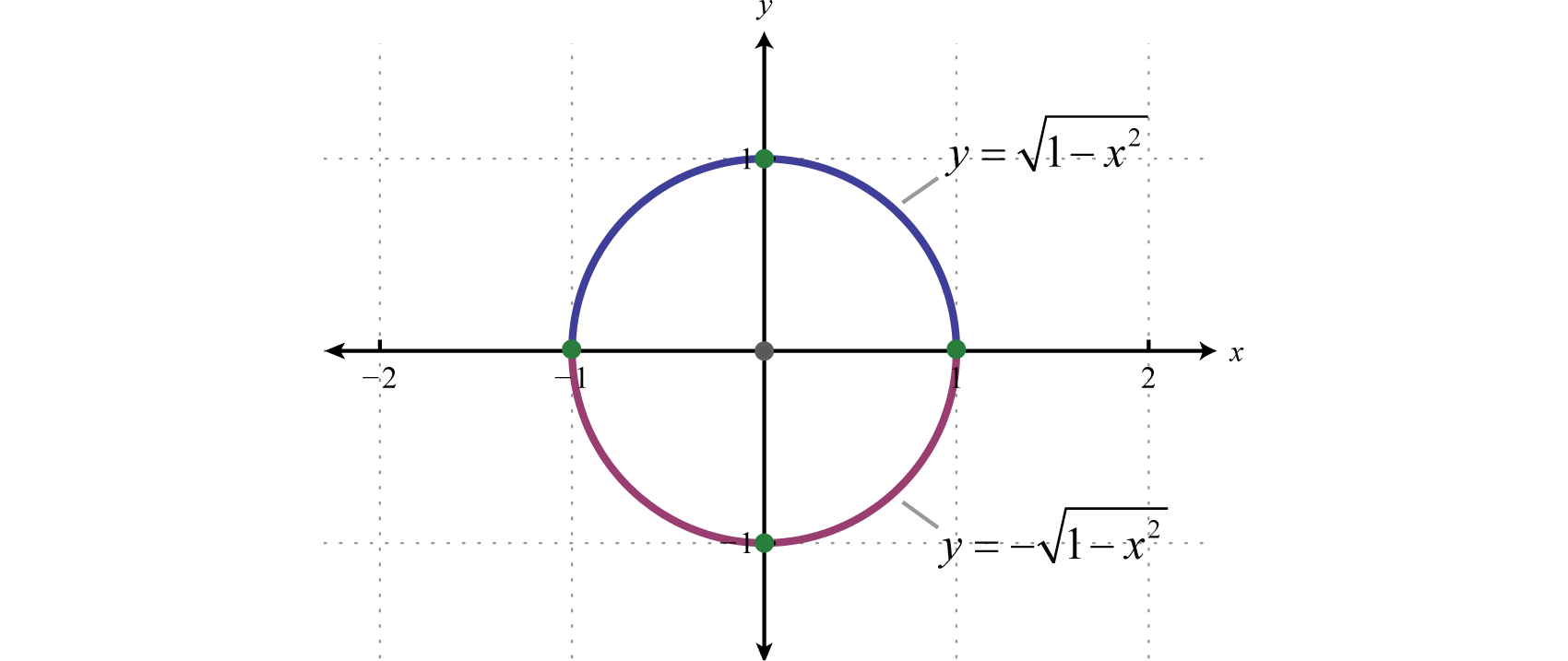
plot\_function(lambda x: np.sin(x))

## 7. Graphing a Circle

* Let's try to graph the unit circle
  + Equation: 
* This cannot be represented as one function
  + We have multiple values of 



* We can try two functions:
  + But we want to represent the circle as one object



def plot\_function(f, x\_min = -10, x\_max = 10, n\_values = 2000):

plt.gca().set\_aspect("equal")

x = np.linspace(x\_min, x\_max, n\_values)

y = f(x)

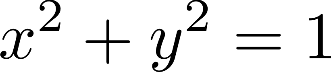
plt.plot(x, y)

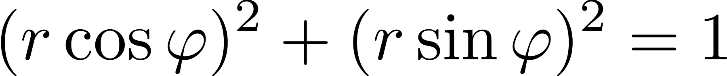
plot\_function(lambda x: np.sqrt(1 - x\*\*2), -1, 1)

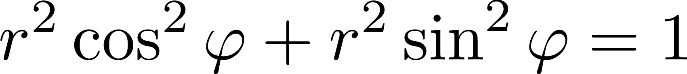
plot\_function(lambda x: -np.sqrt(1 - x\*\*2), -1, 1)

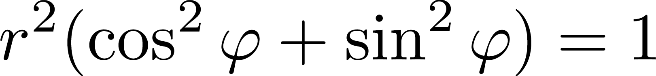
plt.show()

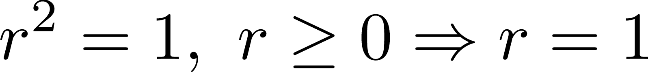
* In math and science, many problems can be solved by **changing  
  our viewpoint**
* We can use another type of **reference system**
  + One which incorporates **angles naturally**
  + **Polar coordinate system :** 
    - (distance from origin (; angle to x-axis)
  + We can easily convert **Cartesian to polar coordinates**











* + Now we can see the equation is very, very simple
  + Doesn't even depend on
  + This is why we needed the change of viewpoint (coordinates)
* Graphing a function in **polar coordinates**
  + This applies to any function, circles in particular
  + Generate initial values of and
  + Convert them to rectangular coordinates
  + Plot the rectangular coordinates

import numpy as np

import matplotlib.pyplot as plt

r = 1 # Radius

phi = np.linspace(0, 2 \* np.pi, 1000) # Angle (full circle)

x = r \* np.cos(phi)

y = r \* np.sin(phi)

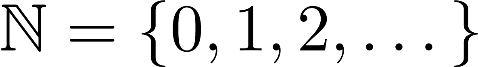
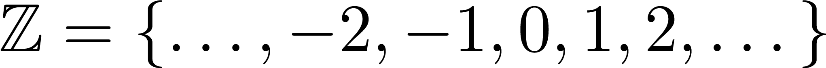
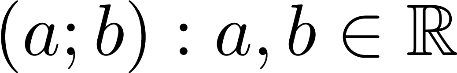
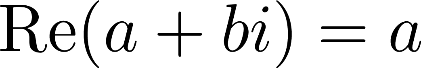
plt.plot(x, y)

plt.gca().set\_aspect("equal")

plt.show()

plt.polar(phi, r)

## 8. Complex Numbers

* Field
  + A collection of values with operations "plus" and "times"
  + Algebra is so abstract we can redefine these operations
* History of number fields
  + **Natural numbers**: 
  + **Integers:** 
  + **Rational numbers  :** ratio of two integers
  + **Real numbers**: 
  + **Complex numbers**: 
    - "Imaginary unit":  is the positive solution of 
    - Pairs of real numbers: 
    - Commonly written as: 
    - **Real part**: 
    - **Imaginary part**: 
    - **In Python, we use j instead of i**

3 + 2j

1j

3j

* + - We can get the real and imaginary parts

z = 3 + 2j

print(z.real) # 3

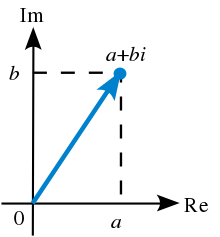
print(z.imag) # 2

* + - Adding and multiplying complex numbers

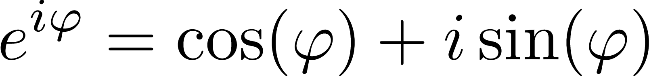
print((3 + 2j) + (8 - 3j)) # (11-1j)

print((3 + 2j) \* (8 - 3j)) # (30+7j)

* + We can plot the coordinate **pairs on the plane**
  + Each point in the **2D space represents one complex number**
  + **Polar coordinates**: we can use the same transformation
    - – **module** of the complex number
    - – **argument** of the complex number
  + Why do we do this?
    - Some operations (e.g., multiplication and division) are easier in polar coordinates
    - Powers of complex numbers become extremely easy
  + Polar form:



## 9. Euler's Formula

* Leonhard Euler proved that: 
  + Here's a [summary of the proof](http://mathworld.wolfram.com/EulerFormula.html)
  + It involves series which **we haven't covered yet**
  + A very beautiful consequence: 
* Now we can write our complex number as: 
* Why and how does multiplication work?
  + Multiplication by a real number
    - Scales the original vector
  + Multiplication by an imaginary number
    - Rotates the original vector
  + You can see a thorough explanation [here](https://betterexplained.com/articles/understanding-why-complex-multiplication-works/)
* **Main point:** Multiplication of complex numbers is the same as scaling and rotating 2D vectors

**10. Fundamental Theorem of Algebra**

* Theorem of Algebra: "**Every non-zero, single-variable, degree- polynomial with complex coefficients has, counted with multiplicity, exactly complex roots**."
* More simply said: **Еvery algebraic equation has as many roots as its power.**
* Back to quadratic equations
  + How do we get all roots?
  + Simply use the complex math Python module: **cmath**

import cmath

def solve\_quadratic\_equation(a, b, c):

discriminant = cmath.sqrt(b \* b - 4 \* a \* c)

return [

(-b + discriminant) / (2 \* a),

(-b - discriminant) / (2 \* a)]

print(solve\_quadratic\_equation(1, -3, -4))

# [(4+0j), (-1+0j)]

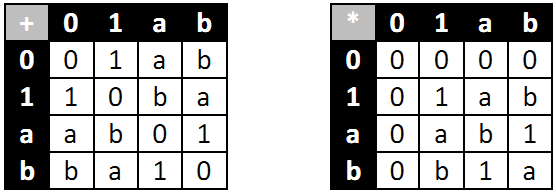
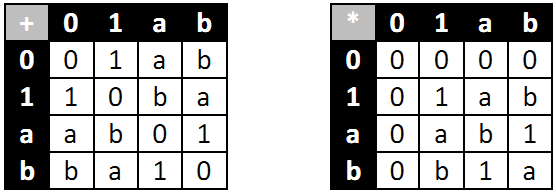
print(solve\_quadratic\_equation(1, 0, -4)) # [(2+0j), (-2+0j)]

print(solve\_quadratic\_equation(1, 2, 1)) # [(-1+0j), (-1+0j)]

print(solve\_quadratic\_equation(1, 4, 5)) # [(-2+1j), (-2-1j)]

**11. Galois Field**

* In everyday algebra, we usually think about fields as those we already know
* But since algebra is abstract, we can define our own fields
* **Galois field:**



* + **Elements**
  + **Addition: equivalent to XOR**
  + **Multiplication: as usual**
* **Usage: in cryptography**
* If you're interested, you can have a look at [this](https://sites.math.washington.edu/~morrow/336_12/papers/juan.pdf) paper