基础数学

顾立平

01数论

- **自然数**：数字1、2、3、4、5……等等。只有正数是包括在这里，它们是已知的最早的系统。
- **整数（Whole numbers）**：加上自然数，“0”的概念后来被接受。
- **整数（Integer）**：包括正自然数和负自然数以及0。
- **有理数**：任何可以表示为分数的数字，如2/3，都是有理数。包括所有有限小数和整数。
- **无理数**：无理数不能表示为分数，包括著名的π、 某些数字的平方根，如2和Euler数e等。
- **实数**：包括有理数和无理数。在数据科学工作中，可将任何小数视为实数。
- **复数和虚数**：取负数的平方根时，会遇到这种数字类型数字。而虚数和复数在某些类型中具有相关性，我们大多会避开它们。

02操作顺序

- 在Python中求解表达式

```
my_value=2*（3+2）**2/5-4
print（my_value）#prints 6.0
```

- 在Python中使用括号以保持清晰

```
my_value=2*（（3+2）**2/5）-4
print（my_value）#prints 6.0
```

- 在编写或者更改代码时，需要注意括号的操作顺序。

03变量

- 在数学中，变量是命名占位符未指定或未知的数字。可以有一个表示任何实数的变量x，可以将其相乘变量，而不声明它是什么。
- Python中的一个变量，然后被乘以三。

```
x = int(input("Please input a number\n"))

product = 3 * x

print(product)
```

- 在数学中，使用θ和β表示线性回归中参数的角度。在Python和R中，我们可能会将这些命名为变量theta和beta等，并且变量名X1可以下标X_1表示，以便产生多个实例X_1, X_2, X_3...X_n等，视情况作为单独的变量。

04函数

x	2x + 1	y
0	2(0) + 1	1
1	2(1) + 1	3
2	2(2) + 1	5
3	2(3) + 1	7

- 函数是定义两个或多个变量之间关系的表达式。
- 函数接受输入变量（也称为域变量或自变量），将它们插入表达式中，然后生成输出变量（也称为因变量）。
- 简单的线性函数：
- $y = 2x + 1$
- 函数很有用，因为它们为变量之间的可预测关系建模，例如，在x温度下，我们可以预期到发生多少次火灾。

04函数

x	2x + 1	y
0.0	2(0) + 1	1
0.5	2(.5) + 1	2
1.0	2(1) + 1	3
1.5	2(1.5) + 1	4
2.0	2(2) + 1	5
2.5	2(2.5) + 1	6
3.0	2(3) + 1	7

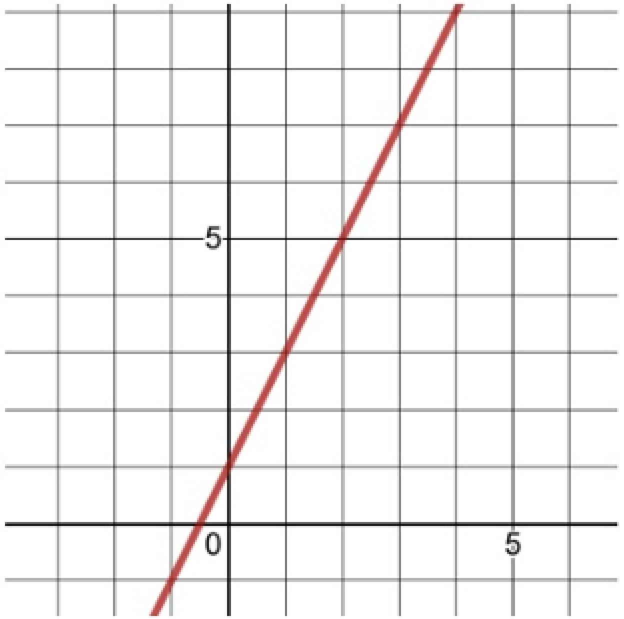
- 因变量y的另一个约定是显式标记它x的函数，例如f(x)。因此函数为y=2x+1可以表示为：

  f(x)=2x+1

- 在Python中声明线性函数

```
def f(x):
    return 2 * x + 1
x_values = [0, 1, 2, 3]
for x in x_values:
    y = f(x)
    print(y)
```

04函数

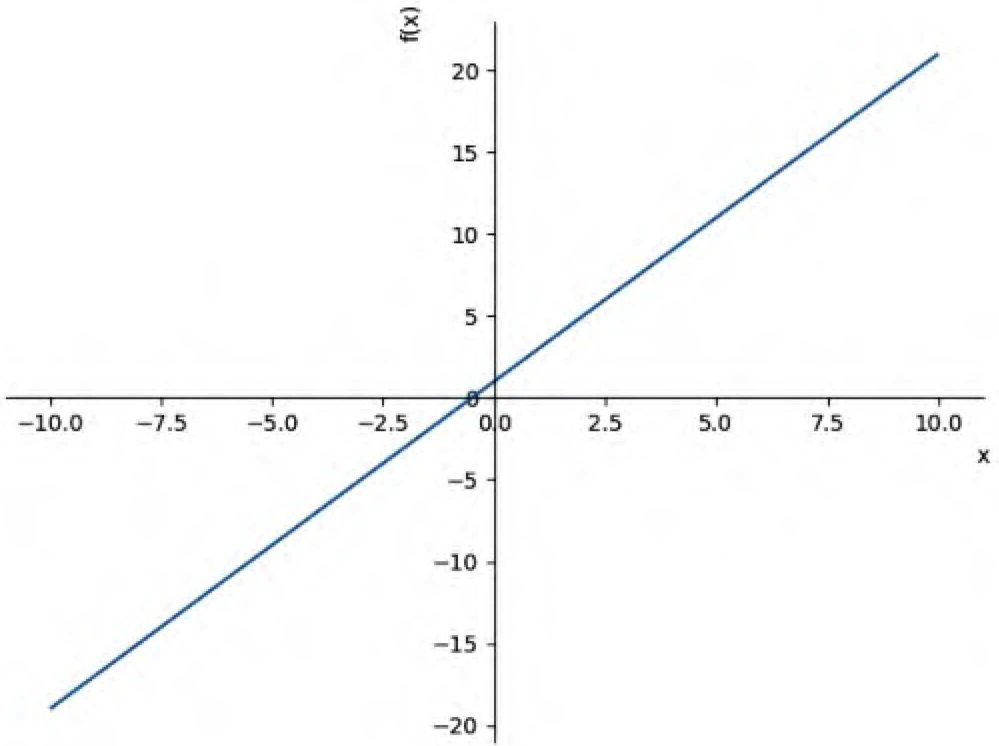



```
from sympy import *
x = symbols('x')
f = 2*x + 1
plot(f)
```

- 当我们用两条数字线（每条一条）在二维平面上绘图时（变量）它被称为笛卡尔平面、x-y平面或坐标平面。我们追踪一个给定x值，然后查找相应的y值，并绘制交点作为一条线。

- 由于实数（或小数，如果您愿意）的性质，有无限多的x值。这是为什么我们在绘制函数 f（x）时，得到一条连续的直线，其中没有间断。有无限多的点在该行或该行的任何部分。

04函数



- **Python库SymPy使用matplotlib绘图，接着只需使用symbols（ ）将x变量声明为函数，接着绘制图形。**

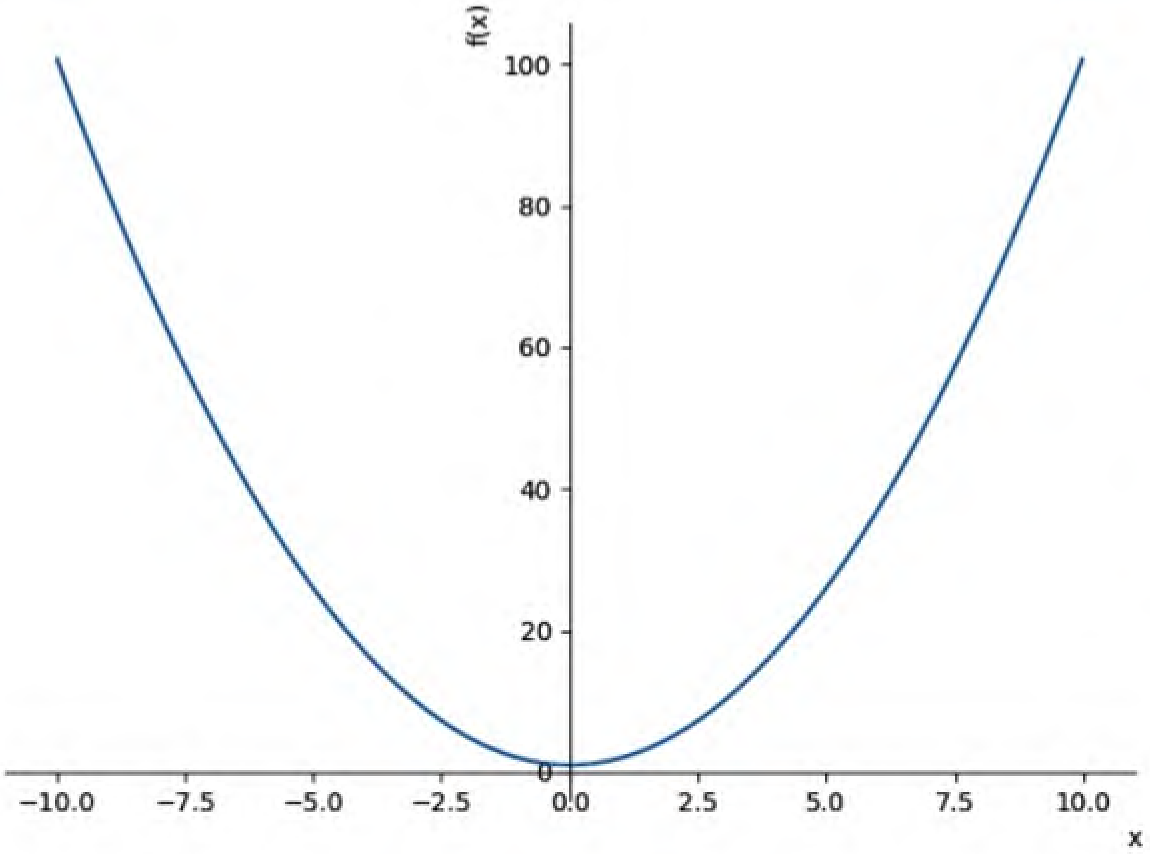
```
from sympy import *

x = symbols('x')

f = 2*x + 1

plot(f)
```

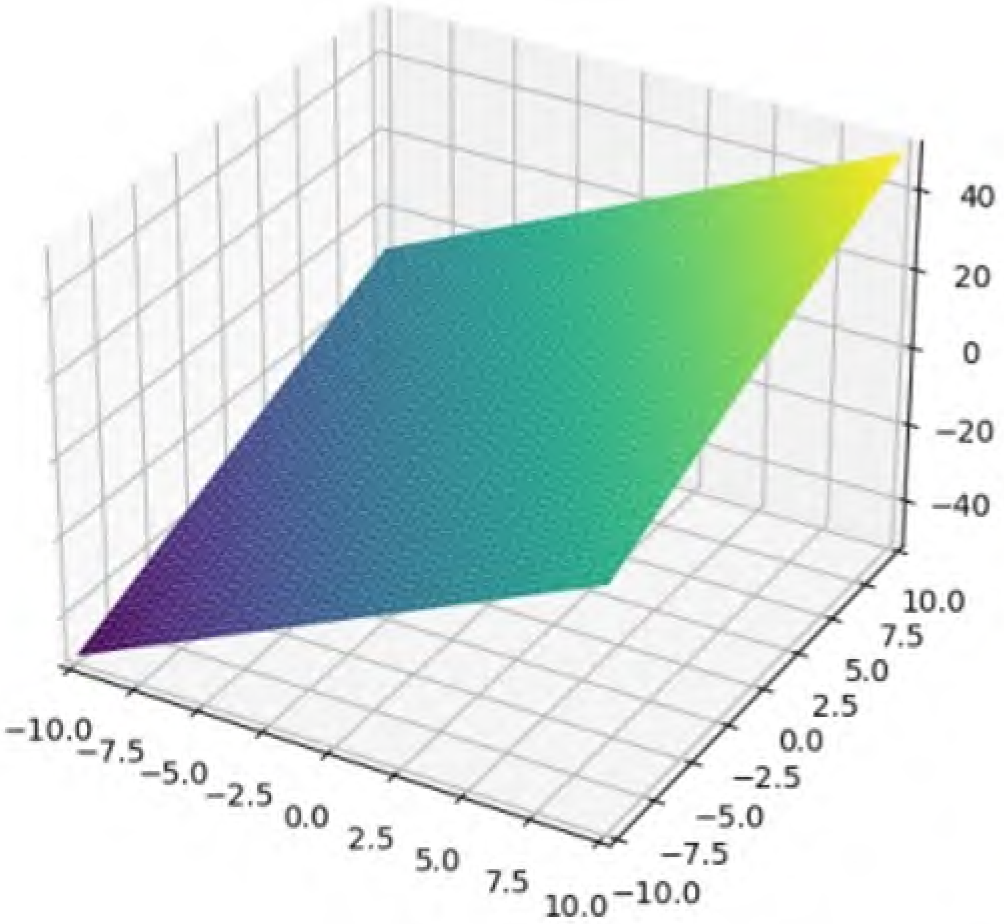
04函数



- 当函数是连续但弯曲的，而不是线性的，称之为曲线函数。

```
from sympy import *
x = symbols('x')
f = x**2 + 1
plot(f)
```

- 此处没有得到一条直线，而是一条平滑、对称的、被称为抛物线的曲线。它是连续的，但不是线性的，它不会产生直线中的值。



04函数

- 当函数使用多个输入变量，而不仅仅是一个时：　f(x，y)=2x+3y
- 因为有两个自变量x和y作为输入，以及一个因变量f（x，y）作为输出，所以我们需要在三维上绘制该图以生成平面，而不是直线。

```
from sympy import *
from sympy.plotting import plot3d
x, y = symbols('x y')
f = 2*x + 3*y
plot3d(f)
```

05加总

- 希腊字母sigma∑表示将元素相加在一起的意思。例如，如果我们想迭代数字1到5，请将每个数字乘以2，然后求和：

$$\sum_{i=1}^{5} 2i = (2)1 + (2)2 + (2)3 + (2)4 + (2)5 = 30$$

- 在Python中执行求和

```
summation = sum(2*i for i in range(1,6))

print(summation)
```

- 函数range（ ）常常用于迭代数据时，在类似x的变量指示集合中的元素，此处用 i 作为一个占位符变量，表示每个连续索引值。

05加总

$$\sum_{i=1}^{n} 10x_i$$

- 通常看到n表示集合中的项目数，如数据集中的记录数。

```
x = [1, 4, 6, 2]
n = len(x)
summation = sum(10*x[i] for i in range(0,n))
print(summation)
```

- 上述，迭代收集大小为n的数，将每个数乘以10，并将它们相加。

- 在Python（以及大多数编程语言）中，我们通常引用从索引0开始的项，而在数学中，我们从索引1开始。因此，我们在迭代中通过从range（ ）中的0开始进行相应的移位。
- 例如，如果调用range（1,4），它将迭代数字1、2和3。它排除了4作为上边界。

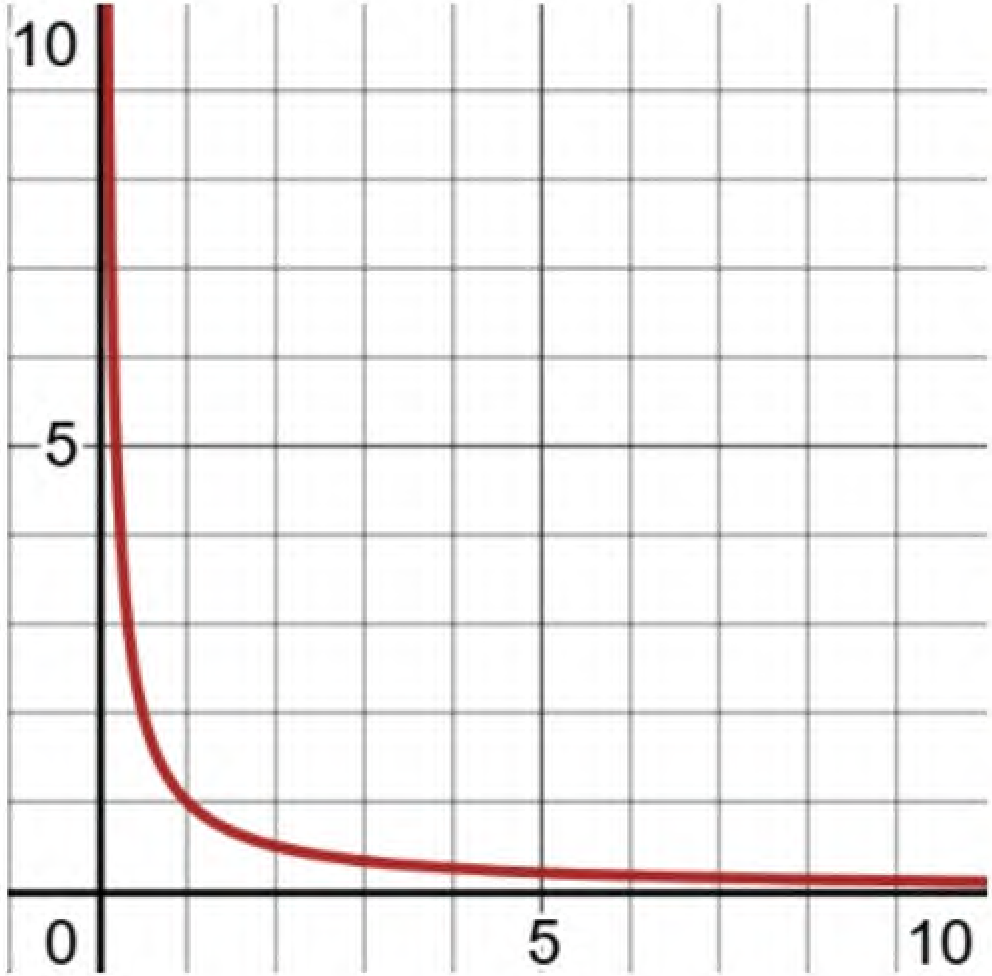
05加总

- 使用Sum（ ）运算符执行SymPy中的求和操作。
- 以下代码，把i从1迭代到n，将每个i相乘，然后求和。
- 使用函数subs（ ）把n指定为5，然后迭代，并求和所有i从1到n的元素：

```
from sympy import *
i,n = symbols('i n')
# 将每个元素i从1迭代到n然后乘法和求和
summation = Sum(2*i,(i,1,n))
# 将n指定为5，迭代数字1到5
up_to_5 = summation.subs(n, 5)
print(up_to_5.doit()) # 30
```

- 在SymPy中的总和是惰性的，这意味着它们不会自动计算或者简化。因此，使用doit（ ）函数来执行表达式。

06指数（Exponents）



指数将数字本身乘以指定的次数。

基数是我们正在求幂的变量或值，指数是乘以基值的次数。

$$\frac{x^2}{x^5} = x^2 \frac{1}{x^5} = x^2 x^{-5} = x^{2+-5} = x^{-3}$$

```
from sympy import *

x = symbols('x')

expr = x**2 / x**5

print(expr) # x**(-3)
```

07对数

$$a^x = b$$

$$log_a b = x$$

- 左侧，把将变量指数重新表示为对数。
- 从代数上讲，这是隔离x的一种方法。

```
from math import log
x = log(8, 2)
print(x) # prints 3.0
```

- 在某些领域，如地震测量，对数的默认基数为10。在数据科学中，对数的默认基数是Euler的数字e这也是Python所使用的。

07对数

运算子	指数属性	对数属性
乘法	$x^m \times x^n = x^{m+n}$	$log(a \times b) = log(a) + log(b)$
除法	$\frac{x^m}{x^n} = x^{m-n}$	$log\left(\frac{a}{b}\right) = log(a) - log(b)$
指数化	$(x^m)^n = x^{mn}$	$log(a^n) = n \times log(a)$
零指数	$x^0 = 1$	$log(1) = 0$
逆向	$x^{-1} = \frac{1}{x}$	$log(x^{-1}) = log\left(\frac{1}{x}\right) = -log(x)$

08欧拉数

$$A = P \times \left(1 + \frac{r}{n}\right)^{nt}$$

- 初始投资P的余额a，利率r，时间跨度t（年数），和时段n（每年的月数）

- 欧拉数e是一个非常类似于Pi（π）的特殊数字，约为2.71828经常在数学上简化了许多问题。
- 欧拉数的指数函数是它自己的导数，这用来处理指数函数和对数函数很方便。在许多应用中，基数（base）并不是真正的物质，我们选择产生最简单导数的那个，那就是欧拉数。
- 这也是为什么它是许多数据科学函数的默认基数。

08欧拉数

$$A = P \times \left(1 + \frac{r}{n}\right)^{nt}$$

- 假设借给某人100元，每年20%的利息。通常，利息将按月复利，每个月的利息是。20/12=。01666。两年后，贷款余额是多少？

$$100 \times \left(1 + \frac{.20}{12}\right)^{12 \times 2} = 148.6914618$$

```
from math import exp

p = 100

r = .20

t = 2.0

n = 12

a = p * (1 + (r/n))**(n * t)

print(a) # prints
148.69146179463576
```

08欧拉数

- **每天复利怎么办？**

$$A = P \times \left(1 + \frac{r}{n}\right)^{nt}$$

$$100 \times \left(1 + \frac{.20}{365}\right)^{365 \times 2} = 149.1661279$$

- **那么...如果继续使这些周期无限小到持续复合，这将导致什么？..**

$$A = P \times \left(1 + \frac{r}{n}\right)^{nt}$$

$$100 \times \left(1 + \frac{.20}{8760}\right)^{8760 \times 2} = 149.1817886$$

08欧拉数

- 欧拉的数字e，大约是2.71828。这里是合成“连续”的公式，意味着我们正在不间断地合成：

$$A = P \times e^{rt}$$

- 从而，计算两年后的贷款余额，如果连续复合，得到：

$$A = P \times e^{rt}$$

$$A = 100 \times e^{.20 \times 2} = 149.1824698$$

08欧拉数

考虑到每分钟的复利，让我们平衡了149.1824584。这使得我们在复利时非常接近149.1824698的价值持续。

```
from math import exp
p = 100 # 本金，起始金额
r = .20 # 年利率，按年计
t = 2.0 # 时间，年数
a = p * exp(r*t)
print(a) # prints 149.1824697641270
```

$$\left(1+\frac{1}{n}\right)^{n}$$

$$\left(1+\frac{1}{100}\right)^{100}=2.70481382942$$

$$\left(1+\frac{1}{1000}\right)^{1000}=2.71692393224$$

$$\left(1+\frac{1}{10000}\right)^{10000}=2.71814592682$$

$$\left(1+\frac{1}{10000000}\right)^{10000000}=2.71828169413$$

08欧拉数

- 在哪里导出这个常数e呢？比较复利公式以及连续利息公式。它们在结构上看起来相似，但有一些差异：

$$A = P \times \left(1 + \frac{r}{n}\right)^{nt}$$

$$A = P \times e^{rt}$$

$$\left(1 + \frac{1}{n}\right)^{n}$$

$$\left(1 + \frac{1}{100}\right)^{100} = 2.70481382942$$

$$\left(1 + \frac{1}{1000}\right)^{1000} = 2.71692393224$$

$$\left(1 + \frac{1}{10000}\right)^{10000} = 2.71814592682$$

$$\left(1 + \frac{1}{10000000}\right)^{10000000} = 2.71828169413$$

- 欧拉数e是表达式 $\left(1 + \frac{1}{n}\right)^{n}$ 的结果值
- 作为n永远变得越来越大，因此接近无穷大。
- 然而，当使n变大时，会有一个递减的回报，并且它大约收敛于在值2.71828上，这是我们的值e。这个e不仅仅用于研究种群及其增长。它在数学的许多领域发挥着关键作用，例如，使用欧拉数来构建正态分布。

09自然对数

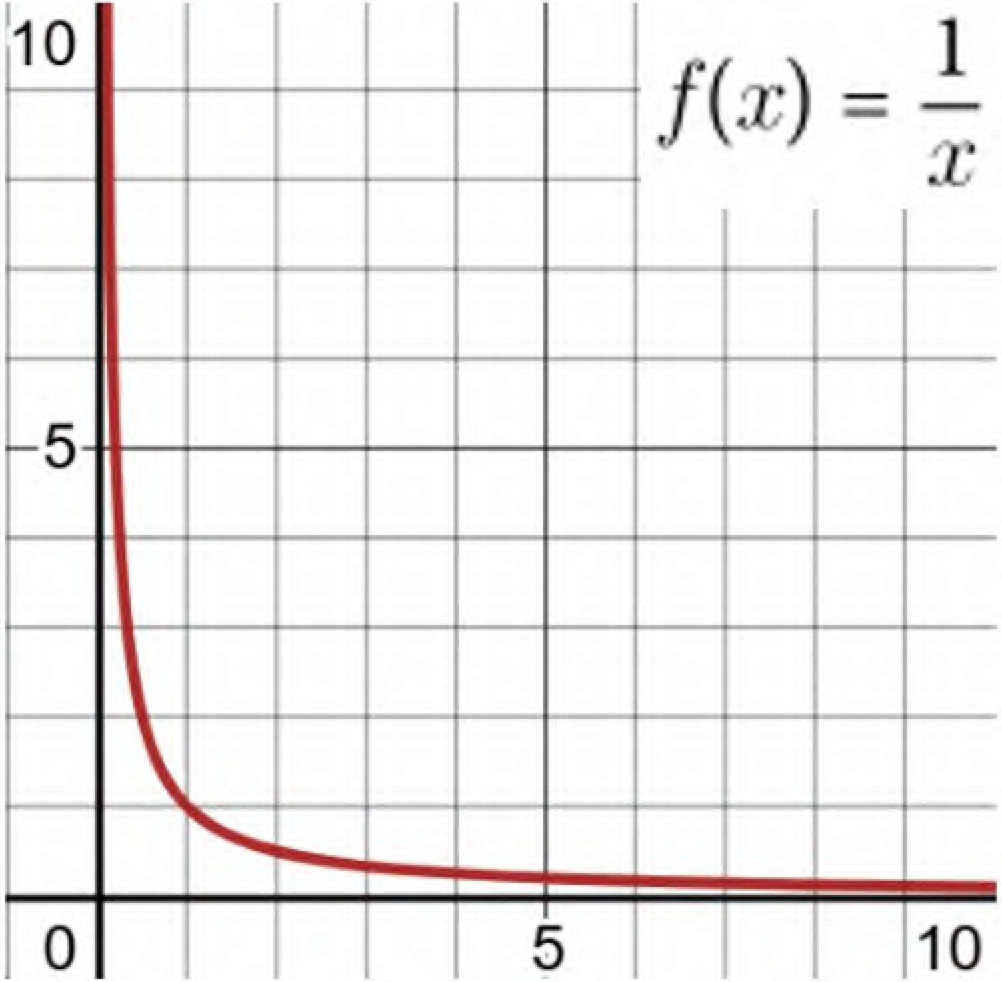
- 当人们用e作为对数的基数时，称它为自然对数。我们可以使用ln（ ）而不是log（ ）来指定自然对数。

```
from math import log
#问e被提升到10的倍数？
x = log(10)
print(x) # prints 2.302585092994046
```

$$\log_e 10 = \ln(10)$$

- 在Python中，自然对数由函数log（ ）指定。不过，函数log（ ）的默认基数是e只需保留第二个参数

10极限



- 左图：我们只看正的x值。当x永远增加时，则f（x）得到接近0。然而f（x）从未真正达到0。它永远都在变更接近。
- 人们表达永恒接近但是从未达到的价值的方式，是通过一个极限：

$$\lim_{x \to \infty} \frac{1}{x} = 0$$

10极限

- 使用SymPy可以计算当x接近无穷大∞时的 $f(x)=\frac{1}{x}$ 值。

```
from sympy import *
x = symbols('x')
f = 1 / x
result = limit(f, x, oo)
print(result) # 0
```

- 这也是发现欧拉数e的方式。它的结果是：对于该函数，将n永远扩展到无穷大：

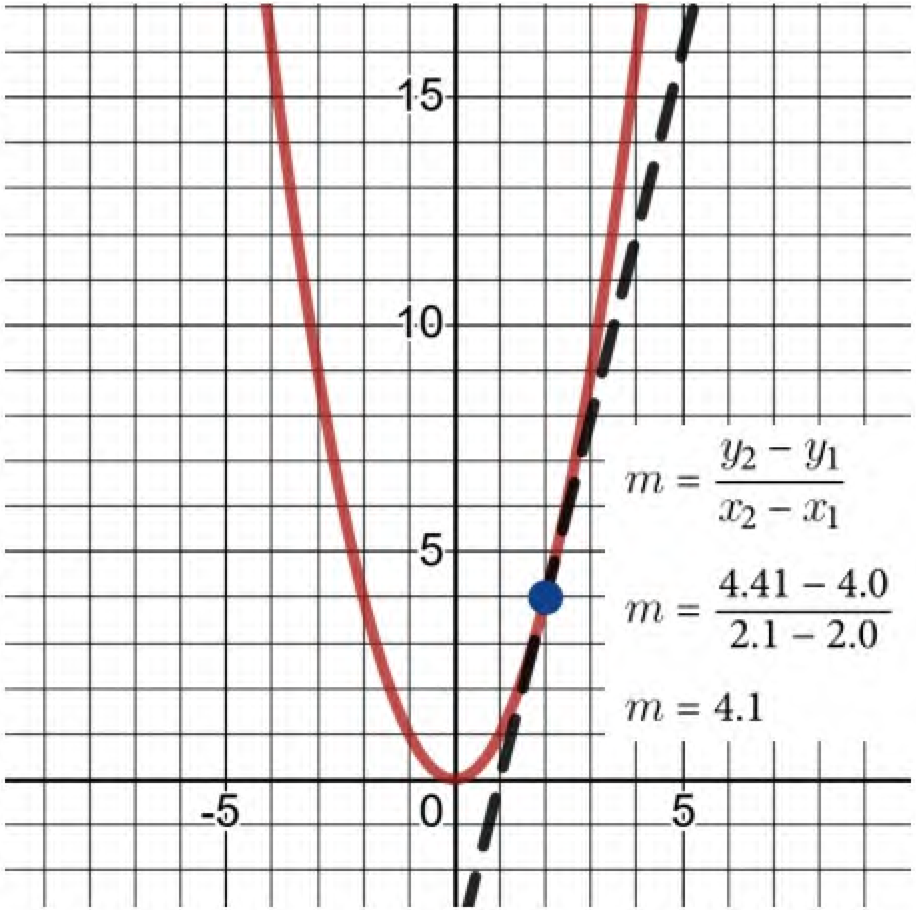
$$\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n = e = 2.71828169413...$$

11导数

- 导数告诉函数的斜率，它用于测量函数中任意点的变化率。它们通常用于机器学习，特别是梯度下降算法。当斜率为0时，这意味着我们处于输出变量的最小值或最大值。
- 当遇到指数函数时，如 $f(x) = x^2$，导数函数将使指数成为乘数，然后将指数递减1，保留导数 $\frac{d}{dx}x^2 = 2x$。表示关于x的导数 $\frac{d}{dx}$ 表示我们正在构建一个以x值为目标的导数，以获得其斜率。为了求x=2处的斜率，因为我们已有导数函数，所以只需x值就可获得斜率：

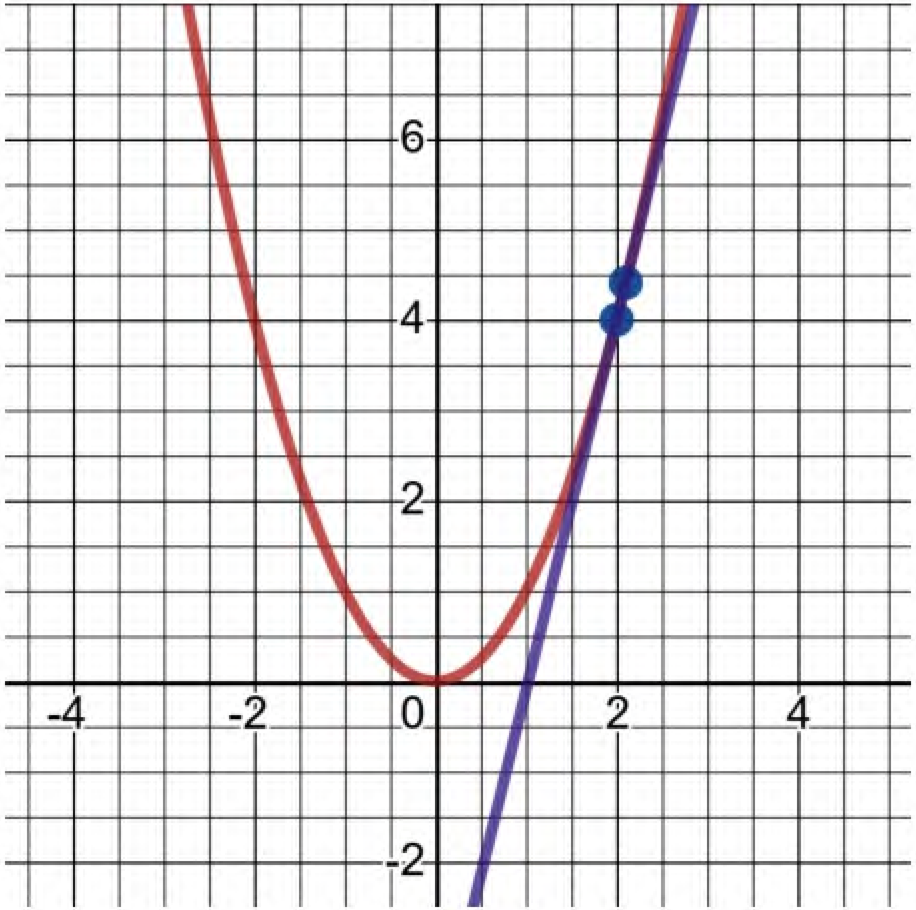
$$f(x) = x^2 \qquad \frac{d}{dx}f(x) = \frac{d}{dx}x^2 = 2x \qquad \frac{d}{dx}f(2) = 2(2) = 4$$

11导数



- 把切线想象为“刚好接触”的直线给定点处的曲线。它还提供给定点的坡度。人们可以粗略地通过创建与给定x值相交的线，来估计给定x值处的切线函数上非常接近的x值。
- 取x=2和附近的值x=2.1，当传递给函数f x=x2时，将产生f（2）=4和f（2.1）=4.41，通过这两点的路径具有4.1的斜率。

11导数



- 如果两点之间的x步长更小，比如x=2和x=2.00001，这将导致

f（2）=4和f（2.00001）=4.00004，这将非常接近实际坡度为4。

- 因此，步长越小到相邻值，我们就越接近得到曲线中给定点的斜率值。

11导数

- 通过在SymPy中使用symbols（ ）函数声明变量，我们可以使用Python语法来声明各种函数，之后使用diff（ ）计算导数函数。
- Python中的导数计算器（不用SymPy而是直接在Python里定义）

```
def derivative_x(f, x, step_size):
    m = (f(x + step_size) - f(x)) / ((x + step_size) - x)
    return m
def my_function(x):
    return x**2
    slope_at_2 = derivative_x(my_function, 2, .00001)
print(slope_at_2) # prints 4.000010000000827
```

$$f(x) = x^2$$

11导数

- 计算导数

```
from sympy import *
# 将“x”声明到SymPy里。
x = symbols('x')
# 现在只需使用Python语法来声明函数
f = x**2
# 计算函数的导数
dx_f = diff(f)
print(dx_f) # prints 2*x
```

$$\frac{d}{dx}x^2 = 2x$$

11导数

- Python中的导数计算器（在SymPy中使用symbols（ ）函数声明变量）

```
def f(x):
        return x**2
def dx_f(x):
        return 2*x
slope_at_2 = dx_f(2.0)
print(slope_at_2)    # prints 4.0
```

$$\frac{d}{dx}f(x) = \frac{d}{dx}x^2 = 2x$$

$$\frac{d}{dx}f(2) = 2(2) = 4$$

- 在SymPy中使用替换功能（调用函数subs（ ）来交换x值为2的变量）

```
# 计算x=2处的斜率
print(dx_f.subs(x,2)) # prints 4
```

12偏导数

$$f(x,y) = 2x^3 + 3y^3$$

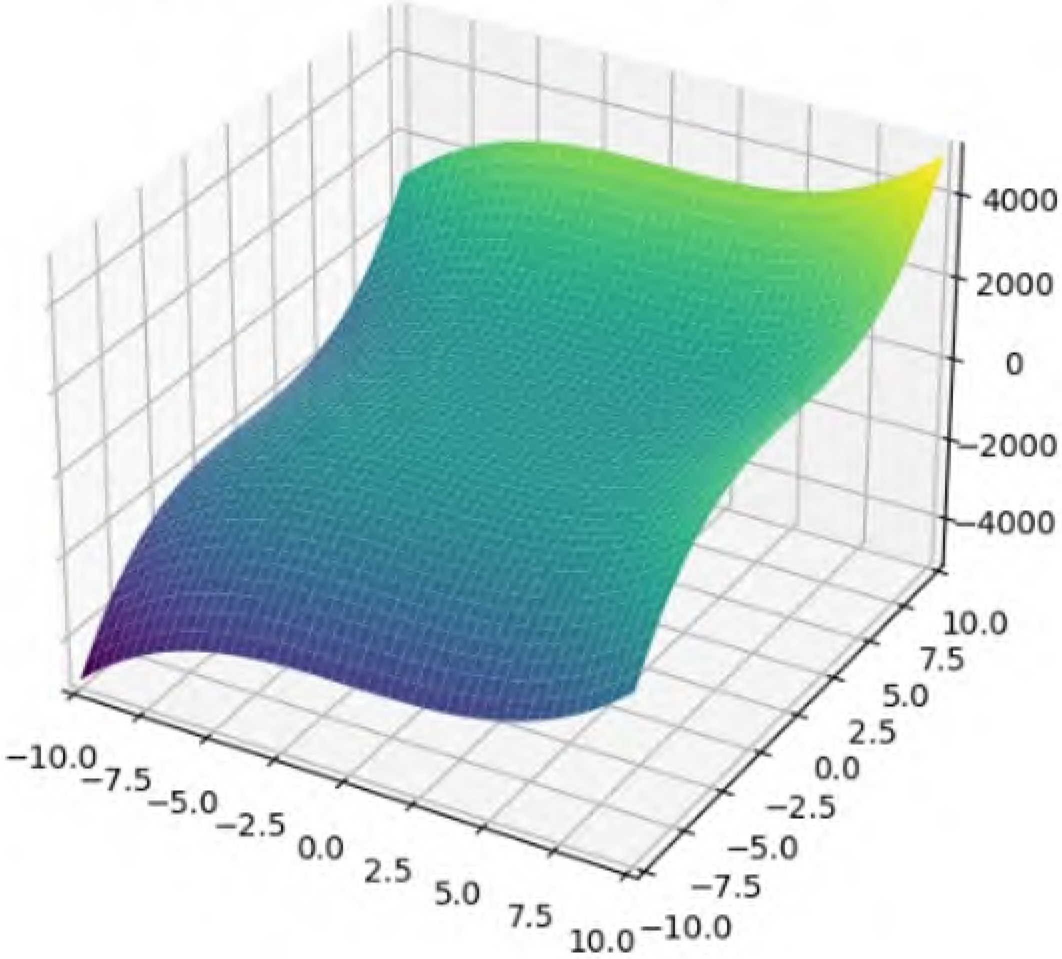
$$\frac{d}{dx}2x^3 + 3y^3 = 6x^2$$

$$\frac{d}{dy}2x^3 + 3y^3 = 9y^2$$

- 在几个方向上，有关于**多个变量的斜率**。对于每个给定的变量导数，我们假设其他变量保持不变。对于两个变量，我们有两个方向的斜率。
- 这d/d_x和d/d_y表示变量X和变量Y的**斜率值**。这些表示相对于多维曲面上每个变量的斜率值。
- 在处理多个维度时，我们在技术上称这些为“**斜率**”**梯度**。当处理多个维度，左侧这些是x和y的导数。

12偏导数

$$f(x, y) = 2x^3 + 3y^3$$




```
from sympy import *
from sympy.plotting import plot3d
# 将x和y声明给SymPy
x,y = symbols('x y')
# 现在只需使用Python语法来声明函数
f = 2*x**3 + 3*y**3
# 计算x和y的偏导数
dx_f = diff(f, x)
dy_f = diff(f, y)
print(dx_f) # prints 6*x**2
print(dy_f) # prints 9*y**2
plot3d(f)
```

12偏导数

$$\lim_{s \to 0} \frac{(x+s)^2 - x^2}{(x+s) - x} \qquad \lim_{s \to 0} \frac{(2+s)^2 - 2^2}{(2+s) - 2} = 4$$

- 通过在闭合处画一条线来近似x=2的斜率；通过添加步长0.0001，相邻点x=2.0001。
- 那么，何不使用极限lim减小步长s，看看它接近的斜率是多少呢？
- 对x=2的斜率感兴趣，因此把左侧式子替换为右侧式子。通过永远将步长s接近0，但永远不会达到它（相邻点不能接触x=2处的点，否则没有线！），收敛到斜率4的极限。

12偏导数

```
from sympy import *
x, s = symbols('x s')
f = x**2
#具有间隙“s”的两点之间的斜率；
#代入溢流量公式
slope_f = (f.subs(x, x + s) - f) / ((x+s) - x)
# 用2代替x的办法
slope_2 = slope_f.subs(x, 2)
#计算x=2处的坡度；无限逼近步长s到0
result = limit(slope_2, s, 0)
print(result) # 4
```

```
from sympy import *
x, s = symbols('x s')
f = x**2
#具有间隙“s”的两点之间的斜率；
#代入溢流量公式
slope_f = (f.subs(x, x + s) - f) / ((x+s) - x)
# 无限逼近步长+s+到0
result = limit(slope_f, s, 0)
print(result) #2
```

13链式法则

- 当组成神经网络层时，会要解开每个层的导数。但是首先，先了解：链式法则。

$$y = x^2 + 1$$

$$z = y^3 - 2$$

- 这两个函数是链接的，因为y是第一个函数，也是第二个函数中的输入变量。这意味着，可以把第一个函数y转换为第二个函数z了。

$$z = \left(x^2 + 1\right)^3 - 2$$

- 那么z对x的导数是多少？我们已经有了替代品：用x来表示z了。

13链式法则

- 求z对x的导数

```
from sympy import *
z = (x**2 + 1)**3 - 2
dz_dx = diff(z, x)
print(dz_dx)
# 6*x*(x**2 + 1)**2
```

因此，z对x的导数是 $6x\left(x^2+1\right)^2$

$$\frac{dz}{dx}\left(\left(x^2+1\right)^3-2\right) = 6x\left(x^2+1\right)^2$$

13链式法则

$$\frac{dz}{dx}\left(\left(x^2+1\right)^3-2\right) = 6x\left(x^2+1\right)^2$$

- 这表示对复合函数 $(x^2+1)^3-2$ 关于 x 的导数。

➢内部函数$u=x^2+1$ 的导数是 $u'=2x$。

➢外部函数$y=u^3-2$ 对 u 的导数是 $y'=3u^2$。。

➢根据链式法则，复合函数的导数等于内部函数导数乘以外部函数对内部函数的导数。

➢因此，复合函数的导数为 $y'\cdot u'=3u^2\cdot 2x=6xu^2$。

➢将$u=x^2+1$ 代入，得到 $6x(x^2+1)^2$。

13链式法则

- 如果我们取导数y和z函数的类型，然后将它们相乘，这也产生z对x的导数！ $$\frac{dy}{dx}\left(x^2+1\right)=2x$$

$$\frac{dz}{dy}\left(y^3-2\right)=3y^2$$

$$\frac{dz}{dx}=(2x)\left(3y^2\right)=6xy^2$$

- $6xy^2$不像$6x(x^2+1)^2$，那是因为我们还没有代入y函数。这样，整个d_z/d_x导数都是用x表示的，没有y

$$\frac{dz}{dx}=6xy^2=6x\left(x^2+1\right)^2$$

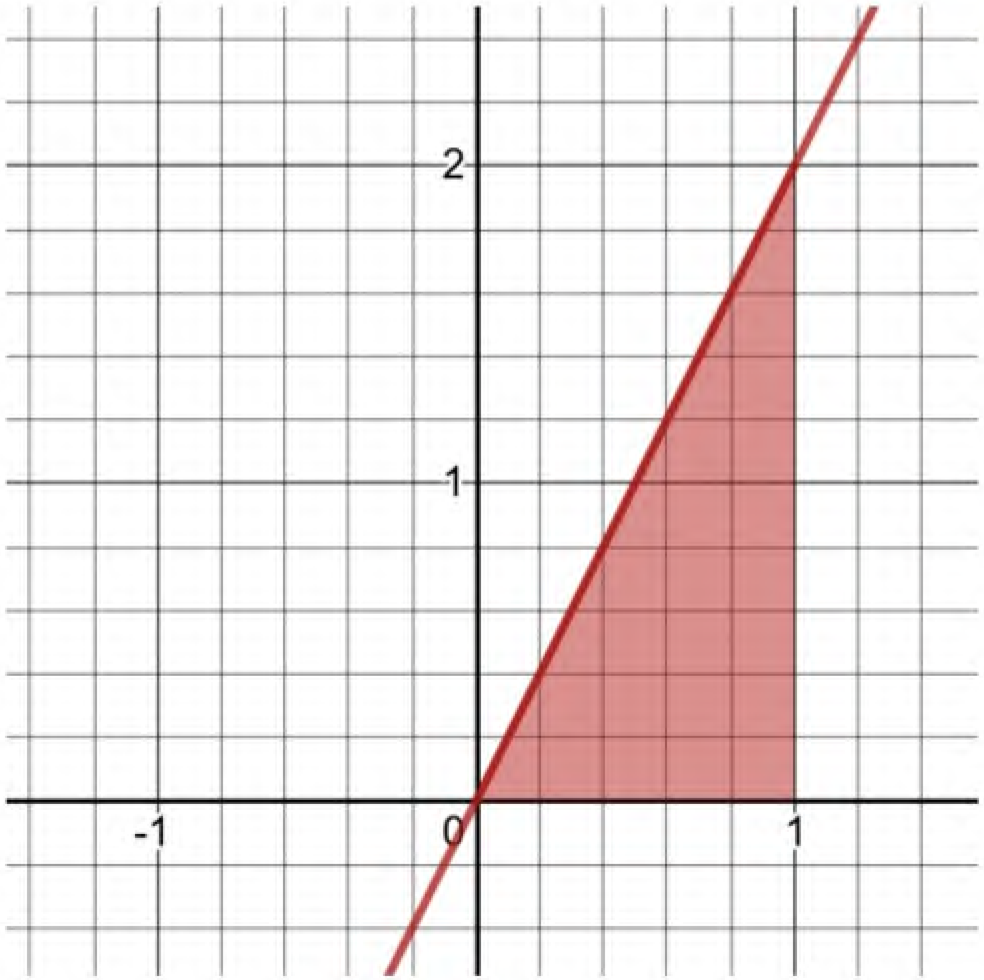
13链式法则

```
from sympy import *
x, y = symbols('x y')
    # 第一函数的导数，需要下划线y以防止变量冲突
_y = x**2 + 1
dy_dx = diff(_y)
    # 第二函数的导数
z = y**3 - 2
dz_dy = diff(z)
    # 计算有无导数；链式法则，代换y函数
dz_dx_chain = (dy_dx * dz_dy).subs(y, _y)
dz_dx_no_chain = diff(z.subs(y, _y))
    # 通过显示两者相等来证明链式法则
print(dz_dx_chain) # 6*x*(x**2 + 1)**2
print(dz_dx_no_chain) # 6*x*(x**2 + 1)**2
```

13链式法则

- 链式规则，它表示对于给定的函数y（具有输入变量x）组合成另一个函数z（输入变量y），人们首先找到导数，通过将两个各自的导数相乘，得到关于x的z：
$$\frac{dz}{dx}=\frac{dz}{dy}\times\frac{dy}{dx}$$
- 链式规则是训练具有适当权重和偏差的神经网络的关键部分。我们可以将每个节点的导数相乘，而不是以嵌套洋葱的方式解开每个节点的衍生物，这在数学上要容易得多。

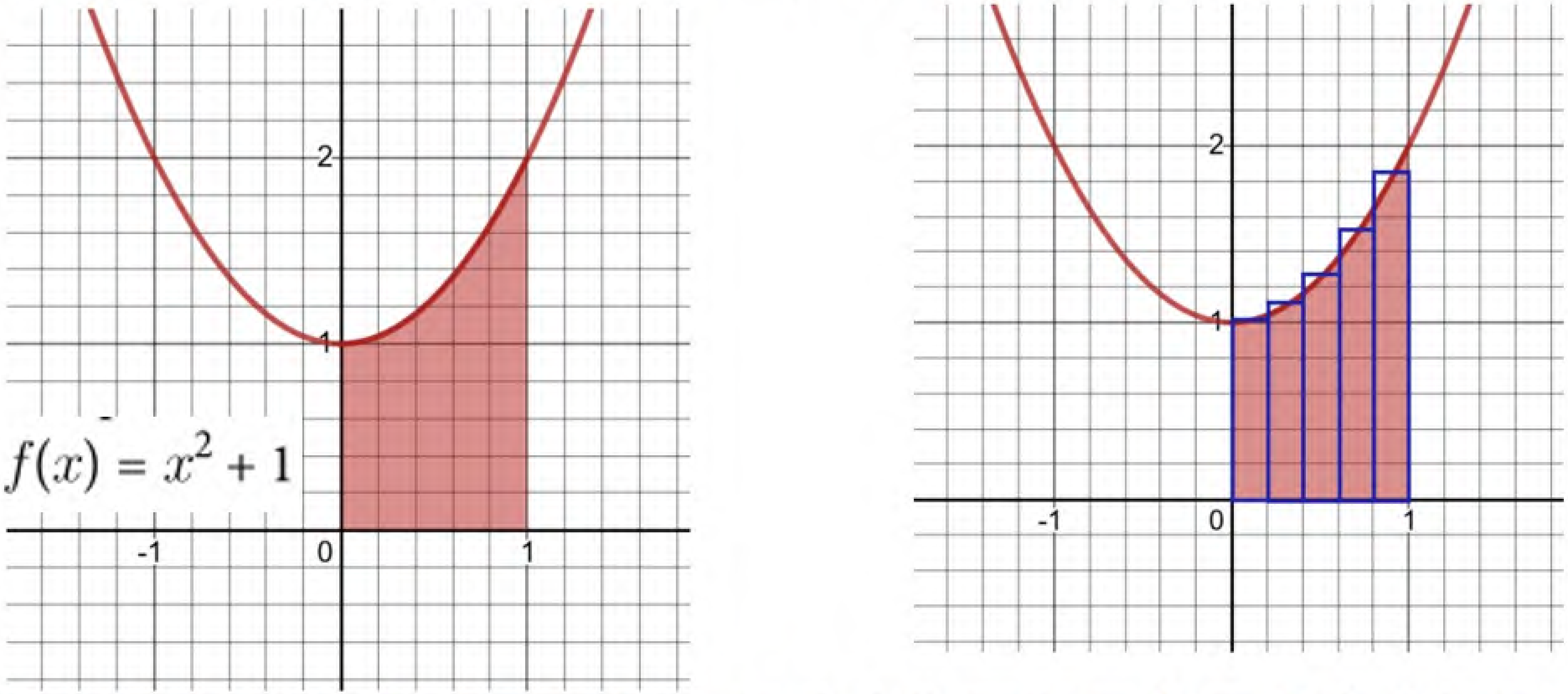
14积分



- 积分用于找到曲线下的面积的给定范围。
- 假设有一个函数fx=2x而我们想找到0和1之间的线下的区域。
- 根据三角形的面积，左图面积可以求出：
- A = 1/2 bh
- A = 1/2 *1*2
- A = 1

14积分

- 那如果是在曲线 f(x) = x2 + 1 下0和1之间的面积是多少呢？



- 矩形的面积为a=长度×宽度，因此我们可以很容易地求出矩形。
- 随着我们增加矩形在减少其宽度的同时，我们会更接近曲线。

14积分

- 上述想法，构成了“积分近似”的内容，代码如下：

```
def approximate_integral(a, b, n, f):
    delta_x = (b - a) / n
    total_sum = 0
    for i in range(1, n + 1):
        midpoint = 0.5 * (2 * a + delta_x * (2 * i - 1))
        total_sum += f(midpoint)
    return total_sum * delta_x
def my_function(x):
    return x**2 + 1
area = approximate_integral(a=0, b=1, n=5, f=my_function)
print(area) # prints 1.33
```

14积分

- 如果使用1000个矩形，代码如下：

```
area = approximate_integral(a=0, b=1, n=1000, f=my_function)
print(area) # prints 1.333333250000001
```

- 得到了更高的精度，得到了更多的小数位。
- 那么，一百万个矩形，如何？

```
area = approximate_integral(a=0, b=1, n=1_000_000, f=my_function)
print(area) # prints 1.3333333333332733
```

- 在这里得到的回报越来越少，并在价值上趋同1.333了，其中“.333”部分永远重复出现。如果这是一个有理数，那就是4/3=1.333。随着矩形数量的增加，近似开始以越来越小的小数达到其极限。

14积分

- 普通Python（和许多编程语言）仅支持小数，但像SymPy这样的计算机代数系统，能够给出精确的有理数。

```
from sympy import *
x = symbols('x')
f = x**2 + 1
area = integrate(f, (x, 0, 1))
print(area) # prints 4/3
```

14积分

将矩形放在曲线下使它们无限小，直到我们接近精确的面积。当然矩形的宽度不能为0。使用极限计算积分：

```
from sympy import *
x, i, n = symbols('x i n')
f = x**2 + 1
lower, upper = 0, 1
delta_x = ((upper - lower) / n) # 每个矩形的长度delta_x
x_i = (lower + delta_x * i) # 每个矩形的起点x_i，其i是每个矩形的索引。
fx_i = f.subs(x, x_i) # 每个矩形的高度fx_i
n_rectangles = Sum(delta_x * fx_i, (i, 1, n)).doit() # 声明n个矩形
area = limit(n_rectangles, n, oo)
print(area) # prints 4/3
```

谢谢！

gulp@mail.las.ac.cn