**离散数学**

**目录**

**[第1章 逻辑 2](#_Toc7357)**

[1.1 命题 2](#_Toc14809)

[1.2 复合命题 6](#_Toc1592)

[1.3 逻辑等价 9](#_Toc19572)

[1.4 谓词和量词 12](#_Toc29800)

[1.5 证明 15](#_Toc22144)

[1.6 布尔代数 16](#_Toc1166)

[1.7 逻辑门电路 20](#_Toc5838)

**[第2章 集合 22](#_Toc32387)**

[2.1 集合 22](#_Toc20726)

[2.2 集合运算 25](#_Toc19990)

[2.3 集合恒等式 27](#_Toc1468)

[2.4 笛卡尔积 29](#_Toc27750)

**[第3章 函数 31](#_Toc32647)**

[3.1 函数的概念 31](#_Toc24825)

[3.2 上取整/下取整函数 33](#_Toc23444)

[3.3 函数的性质 34](#_Toc160)

[3.4 反函数 36](#_Toc15826)

[3.5 合成函数 37](#_Toc107)

[3.6 指数函数/对数函数 38](#_Toc3206)

**[第4章 数论 40](#_Toc25905)**

[4.1 进制转换 40](#_Toc14890)

[4.2 素数 43](#_Toc23682)

[4.3 序列 45](#_Toc5922)

[4.4 递推关系 47](#_Toc12794)

[4.5 求和 50](#_Toc12860)

[4.6 数学归纳法 51](#_Toc27720)

**第1章 逻辑**

**1.1 命题**

**命题(Proposition)**

**逻辑**(logic)规则给出数学语句的准确含义，这些规则用来区分有效和无效的数学**论证**。逻辑不仅对理解数学推理十分重要，而且在计算机科学中有许多应用，逻辑可用于**电路设计**、**程序构造**、**程序正确性证明**等方面。

**命题**(proposition)是逻辑的基本成分，一个**命题**是一个具有**真值**(truth value)的语句，命题可以为**真**也可以为**假**，但不能既为真又为假。

|  |  |
| --- | --- |
| **命题** | **非命题** |
| I have a dog.  1 + 2 = 3  Today is Wednesday.  It is snowing today. | What day is today?  Shut the door!  1 + 2  x + 1 = 2 |

命题习惯上用字母p、q、r、s等来表示，如果一个命题是**真命题**，它的真值为真，用“**T**”表示；如果一个命题是**假命题**，它的真值为假，用“**F**”表示。

**非运算符(Negation Operator / NOT)**

**非运算符**“”只作用于一个命题，其作用是**反转命题的真值**。

**真值表**(truth table)可以给出命题真值之间的关系，在确定由简单命题组成的命题的真值时，真值表特别有用。

|  |  |
| --- | --- |
| **p** | **p** |
| T | F |
| F | T |

|  |
| --- |
| **范例：p** |
| p: It snowed last night.  q: 2 + 3 = 6  p: It didn’t snow last night.  q: |

**合取运算符(Conjunction Operator / AND)**

命题表示“**p并且q**”，当p和q都为真时命题为真，否则为假。真值表如下：

|  |  |  |
| --- | --- | --- |
| **p** | **q** |  |
| T | T | T |
| T | F | F |
| F | T | F |
| F | F | F |

|  |
| --- |
| **范例：** |
| p: 今天是星期五。  q: 今天下雨。  : 今天是星期五并且下雨。 |

**析取运算符(Disjunction Operator / OR)**

命题表示“**p或q**”，当p和q都为假时命题为假，否则为真。真值表如下：

|  |  |  |
| --- | --- | --- |
| **p** | **q** |  |
| T | T | T |
| T | F | T |
| F | T | T |
| F | F | F |

|  |
| --- |
| **范例：** |
| p: 开关坏了。  q: 灯泡坏了。  : 开关坏了或者灯泡坏了。 |

**异或运算符(Exclusive Or / XOR)**

命题表示p和q的**异或**，当p和q种恰有一个为真时命题为真，否则为假。真值表如下：

|  |  |  |
| --- | --- | --- |
| **p** | **q** |  |
| T | T | F |
| T | F | T |
| F | T | T |
| F | F | F |

|  |
| --- |
| **范例：** |
| p: 他现在在上海。  q: 他现在在北京。  : 他现在在上海或北京。 |

|  |  |
| --- | --- |
| **范例：**某地发生了一件谋杀案，警察通过排查确定杀人凶手必为4个嫌疑犯的一个。以下为4个嫌疑犯的供词。  A说：不是我。  B说：是C。  C说：是D。  D说：C在胡说  已知3个人说了真话，1个人说的是假话。  现在请根据这些信息，确定到底谁是凶手。 | |
| *def* main():      for killer in ['A', 'B', 'C', 'D']:          if (killer != 'A') + (killer == 'C') \              + (killer == 'D') + (killer != 'D') == 3:              print(killer)  if \_\_name\_\_ == "\_\_main\_\_":      main() | |
| **运行结果** | C |

**1.2 复合命题**

**复合命题(Compound Proposition)**

使用非运算符和已定义的各联结词可以构造**复合命题**。小括号用于规定复合命题中多个逻辑运算符的操作顺序，为了减少所需的小括号数量，规定了各联结词的**优先级**。

|  |  |
| --- | --- |
| **优先级** | |
| 1 |  |
| 2 |  |
| 3 |  |

**蕴含运算符(Implication Operator)**

命题表示**p蕴含q**，在p为真而q为假时命题为假，否则为真。其中**p称为前提**，**q称为结论**。真值表如下：

|  |  |  |
| --- | --- | --- |
| **p** | **q** |  |
| T | T | T |
| T | F | F |
| F | T | T |
| F | F | T |

表示的术语有很多种，常见的有：

* if p, then q.
* p only if q.
* q is necessary for p.

|  |
| --- |
| **范例：** |
| p: 我去看电影。  q: 我买奶茶。  : 如果我去看电影，那么我会买奶茶。 |



由可以构造出几个相关的蕴含：

1. 称为的**逆命题**(converse)。
2. 称为的**倒置命题**(contrapositive)。

|  |
| --- |
| **范例：逆命题与倒置命题** |
| p: 今天是星期四。  q: 我今天有考试。  : 如果今天是星期四，那么我今天有考试。  ：如果我今天有考试，那么今天是星期四。  ：如果我今天没有考试，那么今天不是星期四。 |

**双蕴含(Biconditional Operation)**

命题表示**p双向蕴含q**，在p和q有相同的真值时命题为真，否则为假。真值表如下：

|  |  |  |
| --- | --- | --- |
| **p** | **q** |  |
| T | T | T |
| T | F | F |
| F | T | F |
| F | F | T |

双蕴含的真值表与异或的真值表正好相反，因此与相同。

**1.3 逻辑等价**

**逻辑等价(Logical Equivalence)**

两个不同的复合命题可能拥有完全相同的真值，则称这两个命题在逻辑上是**等价**的。如果无论复合命题中各个命题的真值是什么，复合命题的真值**总是为真**，这样的复合命题称为**永真式**(tautology)。如果真值**永远为假**的复合命题称为**矛盾**(contradiction)。

|  |  |  |  |
| --- | --- | --- | --- |
| **p** |  |  |  |
| T | F | T | F |
| F | T | T | F |

如果复合命题s和r是逻辑等价的，可表示为。只有当是永真式时，s和r才是逻辑等价的。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **范例：使用真值表证明** | | | | | | |
| **p** | **q** |  |  |  |  |  |
| T | T | T | F | F | F | T |
| T | F | T | F | T | F | T |
| F | T | T | T | F | F | T |
| F | F | F | T | T | T | F |

一些重要的逻辑等价关系如下：

|  |  |
| --- | --- |
| **名称** | **等价关系** |
| **幂等律**  **(Idempotent Laws)** |  |
| **恒等律**  **(Identity Laws)** |  |
| **支配律**  **(Domination Laws)** |  |
| **双非律**  **(Double Negation Law)** |  |
| **交换律**  **(Commutative Laws)** |  |
| **结合律**  **(Associative Laws)** |  |
| **分配律**  **(Distributive Laws)** |  |
| **德摩根律**  **(De Morgan’s Laws)** | **q**  **q** |
| **吸收律**  **(Absorption Laws)** |  |
| **条件恒等** |  |

|  |
| --- |
| **范例：证明永真** |
|  |

**1.4 谓词和量词**

**谓词(Predicate)**

命题逻辑并不能表达数学语言和自然语言中所有语句的确切含义。在数学表达式和计算机程序中经常可以看到含有变量的语句，例如“x > 3”、“x = y + 3”、“程序x正在运行”等。当变量值未指定时，这些语句既不为真也不为假。

利用P(x)可以表示语句，其中x是变量，语句P(x)可以说是命题函数P在x的值。一旦给变量x赋一个值，语句P(x)就称为命题并具有真值。

通常使用大写字母P、Q、R等表示**谓词**，小写字母x、y、z等表示**变量**。

|  |  |
| --- | --- |
| **范例：谓词** | |
| P(x): x + 3 = 6，P(3)的真值？ | True |
| Q(x, y): x = y + 2，Q(4, 1)的真值？ | False |

**量词(Quantifier)**

**量词**用量化表示在何种程度上谓词对于一定范围的个体成立。

量词有**全称量词**(universal quantifier)和**存在量词**(existential quantifer)：

1. **全称量词**用“”表示“ALL”。是一个命题，当范围内**所有**的x都能使语句P(x)为真时，命题为真。

|  |
| --- |
| **范例：全称量词** |
| 假设x表示“全班所有学生”，P(x)表示“x完成了作业”。  ：全班所有学生都完成了作业。 |

2. **存在量词**用“”表示“Exist”。是一个命题，当范围内**存在至少一个**x能够语句P(x)为真时，命题为真。

|  |
| --- |
| **范例：存在量词** |
| 假设x表示“全班所有学生”，P(x)表示“x完成了作业”。  ：班里存在有一个学生完成了作业。 |

|  |
| --- |
| **范例：嵌套量词** |
| 假设x表示“一个人”，P(x)表示“x有父母”。  ：所有人都有父母。  ：存在至少有一个人有父母。  ：至少存在一个人x和一个人y有父母。 |

|  |  |
| --- | --- |
| **范例：P(x): x是偶数，Q(x): x能被3整除，。** | |
| 的真值？ | True |
| 的真值？ | False |

**全称量词的否定**

否定运算符可以使用在全称量词上。两个等价关系如下：

1.

2.

|  |
| --- |
| **范例：与** |
| P(x): x will pass the course (x is a student).  : Not all students will pass the course.  : No student will pass the course. |

|  |
| --- |
| **范例：与** |
| P(x): x will pass the course (x is a student).  : There does not exist a student that will pass the course.  : There exists a student that will not pass the course. |

**1.5 证明**

**证明(Proof)**

**证明方法**非常重要，不仅因为它们可用于证明数学定理，而且在计算机科学中也有许多应用，包括验证程序正确性、建立安全的操作系统、人工智能领域做推论等。证明就是建立定理真实性的**有效论证**。

证明定理有很多方法：

1. **直接证明法**(direct proof)

|  |
| --- |
| **范例：直接证明法** |
| 定理：如果n是奇数，那么也是奇数。  当 |

2. **反证法**(proof by contrapositive)：由于，因此可以通过证明原命题的**逆否命题**来反证原命题。

|  |
| --- |
| **范例：反证法** |
| 定理：当，如果x\*y是偶数，那么x是偶数或y是偶数。  逆否命题：当，如果x是奇数并且y是奇数，那么x\*y是奇数。 |

**1.6 布尔代数**

**布尔代数(Boolean Algebra)**

计算机和其它电子设备中的**电路**都有**输入**和**输出**，输入是0或1，**输出也是0或1**。电路可以用任何具有**两个不同状态**的基本元件来构造，例如开关和光学装置就是这样的原件，开关可位于开或关的位置，光学装置可能是点亮或未点亮。18世纪，乔治·布尔（George Boole）给出了逻辑的基本规则。

电路的操作可以用**布尔函数**来定义，这样的**布尔函数对任意一组输入都能指出其输出的值**。

布尔代数提供的是**集合{0, 1}上的运算和规则**，布尔代数的规则类似于命题逻辑的规则。“**1**”相当于逻辑中的“**真**”，“**0**”相当于逻辑中的“**假**”。

布尔代码运算主要有三种：

1. **补**(complement)

|  |  |
| --- | --- |
| **x** |  |
| 1 | 0 |
| 0 | 1 |

2. **布尔积**(boolean multiplication)

|  |  |  |
| --- | --- | --- |
| **x** | **y** |  |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

3. **布尔和**(boolean addition)

|  |  |  |
| --- | --- | --- |
| **x** | **y** | **x + y** |
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

|  |
| --- |
| **范例：布尔表达式** |
| 当x = y = 1，w = z = 0时， |

一些重要的布尔代数关系如下：

|  |  |
| --- | --- |
| **名称** | **等价关系** |
| **幂等律**  **(Idempotent Laws)** |  |
| **恒等律**  **(Identity Laws)** |  |
| **支配律**  **(Domination Laws)** |  |
| **双非律**  **(Double Negation Law)** |  |
| **交换律**  **(Commutative Laws)** |  |
| **结合律**  **(Associative Laws)** |  |
| **分配律**  **(Distributive Laws)** |  |
| **德摩根律**  **(De Morgan’s Laws)** |  |
| **吸收律**  **(Absorption Laws)** |  |

|  |
| --- |
| **范例：证明** |
|  |

**布尔函数(Boolean Function)**

含有n个变量的布尔函数能够构造出2n行的输入输出表。

|  |  |  |  |
| --- | --- | --- | --- |
| **范例：计算布尔函数的值** | | | |
| **x** | **y** | **z** | **F(x, y, z)** |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

**NAND和AND运算符**

**NAND**运算符用“”表示“**Not And**”，。

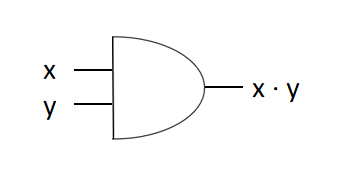
**NOR**运算符用“”表示“**Not Or**”，。

**1.7 逻辑门电路**

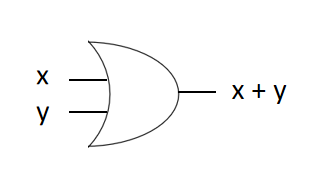
**逻辑门电路(Logical Gate Circuit)**

基础的**逻辑门**主要有三种：

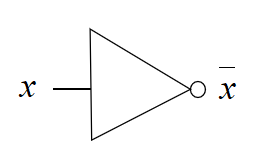
1. **与门**（**AND** gate）：



2. **或门**（**OR** gate）：



3. **非门**（**NOT** gate）：



|  |
| --- |
| **范例：设计一个投票表决电路，三个人中有两人赞成即通过。赞成票为1，否决表为0。布尔函数为F(x, y, z) = xy +xz +yz。** |
|  |

**第2章 集合**

**2.1 集合**

**集合(Set)**

**集合**是对象的**唯一**的、**无序**的聚集，通常一个集合中的对象都具有**相似**的性质。**对象**也称为集合的**元素**（element）或**成员**（member）。

通常用大写字母表示集合，小写字母表示元素。表示a是集合A中的元素，表示a不是集合A中的元素。

使用**花名册方法**（roster method）列出集合中的元素，可以用于描述集合。

|  |
| --- |
| **范例：花名册方法** |
| 小写元音字母集合V = {a, e, i, o, u}  小于10的正奇数集合O = {1, 3, 5, 7, 9}  小于100的非负整数集合A = {0, 1, 2, 3, ..., 99} |

**集合构造器**（set builder）通过描述元素具有的形式来描述集合。

|  |
| --- |
| **范例：集合构造器** |
| 小于10的正整数A = {x | x < 10} |

有一些常用的特殊符号可用于描述指定的集合：

|  |  |
| --- | --- |
| **符号** | **含义** |
|  | 自然数集{0, 1, 2, 3, ...} |
|  | 整数集{..., -2, -1, 0, 1, 2, ...} |
|  | 正整数集{1, 2, 3, ...} |
|  | 有理数集 |
|  | 正有理数集 |
|  | 实数集 |
|  | 正实数集 |
|  | 复数集 |
|  | 空集{} |

**基数(Cardinality)**

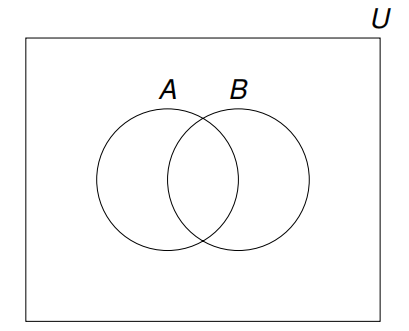
**基数**表示**有限集合**中元素的**个数**，集合A的基数记为“|A|”。

|  |
| --- |
| **范例：基数** |
| 英语字母集合A，|A| = 26  空集， |

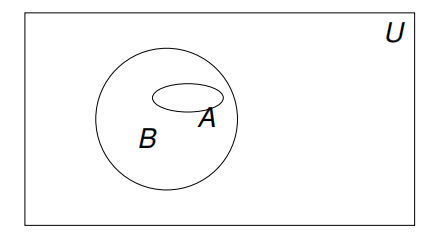
**文氏图(Venn Diagram)**

集合还可以使用**文氏图**来表示。

**全集**(universal set)包含所研究问题中**所有的元素**，用符号“”表示。



假设有两个集合A和B，如果A中的所有元素都B中，那么A就是B的子集，表示为“”。如果A中有一个元素不在B中，那么A就不是B的子集，表示为“”。



**只有当两个集合互相为对方的子集时，那么这两个集合相等**，即：

如果，并且B中有一个元素不是A的元素，那么称A是B的**真子集**(proper subset)，表示为“”。

**幂集(Power Set)**

一个集合中是可以包含另一个集合的，如。需要注意，。

**幂集**用于表示**一个集合所有子集的集合**，集合A的幂集表示为P(A)。

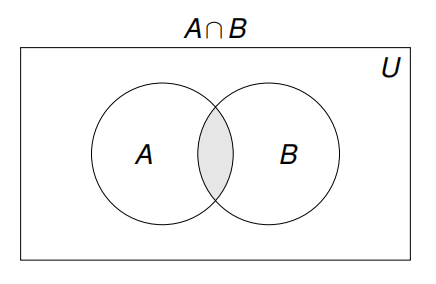
|  |
| --- |
| **范例：幂集** |
|  |

如果集合A的基数为n，那么A的幂集的基数为，即。

**2.2 集合运算**

**交集(Intersection)**

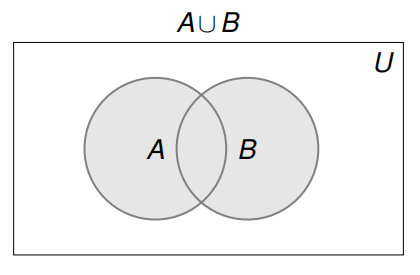
假设A和B是两个集合，由所有属于A并且属于B的元素所组成的集合，称为A与B的**交集**，表示为“”。



如果两个集合没有公共元素，那么它们的交集为空集。

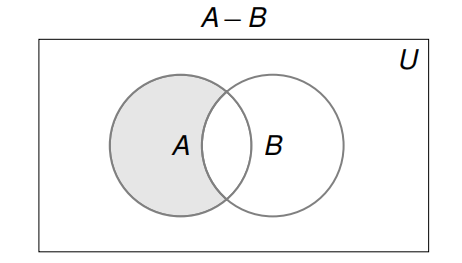
**并集(Union)**

假设A和B是两个集合，由它们所有元素合并在一起组成的集合，称为A与B的**并集**，表示为“”。



**差集(Difference)**

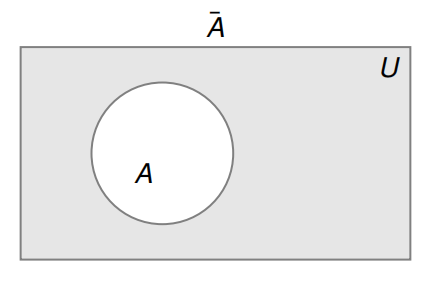
假设A和B是两个集合，由属于A而不属于B的元素组成的集合，称为A与B的**差集**，表示为“A - B”。



**差集运算不满足交换律**，即。

**补集(Complement)**

假设A是一个集合，由全集中所有不属于A的元素组成的集合，称为A的**补集**，表示为“”。



**2.3 集合恒等式**

**集合恒等式**

集合恒等式可以直接由对应的逻辑等价式证明。

|  |  |
| --- | --- |
| **名称** | **等价关系** |
| **幂等律**  **(Idempotent Laws)** |  |
| **恒等律**  **(Identity Laws)** |  |
| **支配律**  **(Domination Laws)** |  |
| **补律**  **(Complement Law)** |  |
| **交换律**  **(Commutative Laws)** |  |
| **结合律**  **(Associative Laws)** |  |
| **分配律**  **(Distributive Laws)** |  |
| **德摩根律**  **(De Morgan’s Laws)** |  |
| **吸收律**  **(Absorption Laws)** |  |

|  |
| --- |
| **范例：证明** |
|  |

|  |
| --- |
| **范例：一共有40个学生，有3门课程可供学生选择（C语言、离散数学、软件工程）。**  **7人没有选任何课程；**  **16人选软件工程；**  **10人选C语言；**  **5人同时选离散数学和软件工程；**  **4人同时选离散数学和C语言；**  **3人同时选软件工程和C语言；**  **2人同时选离散数学、软件工程和C语言。** |
|  |

**2.4 笛卡尔积**

**元组(Tuple)**

有时候元素聚集中**次序**是很重要的，由于集合是**无序**的，所以就需要一种不同的结构表示**有序**的聚集，这就是**有序n元组**(ordered-n-tuple)。

有序n元组是以a1为第1个元素，a2为第2个元素，an为第n个元素的有序聚集。

只有两个有序n元组的**每一对对应的元素都相等**，那么这两个有序n元组是相等的，即：

iff i = 1, 2, ..., n

需要注意，**(a, b)与(b, a)不相等**，**除非a = b**。

**笛卡尔积(Cartesian Product)**

假设有两个集合A和B，A和B的**笛卡尔积**用“”表示，笛卡尔积是**所有序偶(a, b)的集合**，其中且。

|  |
| --- |
| **范例：笛卡尔积的应用** |
| 学生集合A = {s1, s2}  课程集合B = {c1, c2, c3}  笛卡尔积表示学生选课的所有可能情况。 |

**笛卡尔积和是不相等的，除非或或。**

|  |
| --- |
| **范例：笛卡尔积** |
| A = {0, 1}  B = {1, 2}  C = {0, 1, 2} |

一个集合与**自身**的笛卡尔积，如可表示为“”。

|  |
| --- |
| **范例：笛卡尔积** |
| A = {1, 2} |

**第3章 函数**

**3.1 函数的概念**

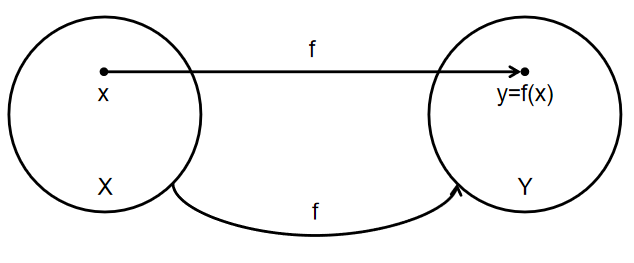
**函数(Function)**

函数在数学和计算机科学中的概念非常重要，在离散数学中函数用于定义像序列和字符串这样的离散结构。

利用一个**函数**f，可以将一个值**映射**(mapping)到一个特定的值y = f(x)上（）。

假设有两个非空集合X和Y，从X到Y的函数f是指对于X的每个元素恰好都**对应**Y的一个元素（），那么就写成。

集合X被称为函数f的**定义域**(domain)，集合Y被称为函数f的**陪域**(co-domain)。如果f(x) = y，那么y是x在函数f下的**像**(image)，x是y载函数f下的**原像**(pre-image)。函数f的**值域**(range)是集合X中所有像的集合。



|  |  |
| --- | --- |
| **范例：判断是否为函数** | |
|  |  |
| 函数 | 不是函数 |

|  |  |
| --- | --- |
| **范例：函数** | |
|  | |
| 是否为函数 | 是 |
| 定义域domain | a, b, c, d |
| 陪域co-domain | 1, 2, 3, 4 |
| 值域range | 2, 3, 4 |

当两个函数f和g有相同的定义域和陪域，并且对于定义域中所有元素x都满足f(x) = g(x)，那么函数f和g相等，表示为“f = g”。

**3.2 上取整/下取整函数**

**上取整函数(Ceiling Function)**

上取整函数和下取整函数可以将实数**映射到整数**（），它们以不同的方式将实数近似到相邻的整数。

**上取整函数将实数x向上取到大于或等于x的最小整数，表示为“”。**

|  |
| --- |
| **范例：上取整函数** |
|  |

**下取整函数(Floor Function)**

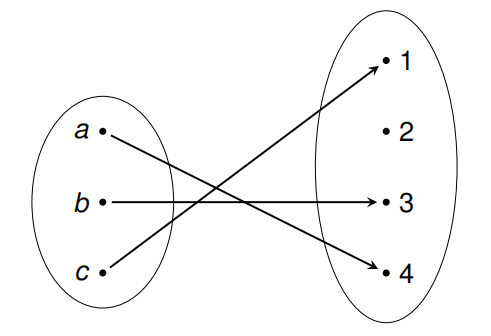
**下取整函数将实数x向下取到小于或等于x的最大整数，表示为“”。**

|  |
| --- |
| **范例：下取整函数** |
|  |

**3.3 函数的性质**

**一对一函数(One-to-one)/单射函数(Injection)**

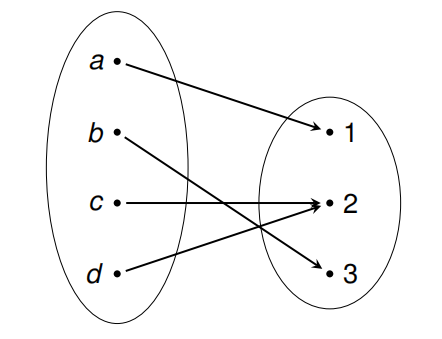
**一对一函数/单射函数**是指对于函数f的定义域中所有的a和b，如果，那么。



|  |
| --- |
| **范例：一对一函数/单射函数** |
| f(x) = x + 1是一对一函数。  f(x) = x2不是一对一函数，因为f(1) = f(-1) = 1。 |

**映上函数(Onto)/满射函数(Surjection)**

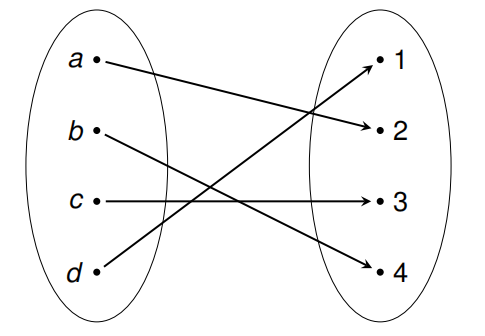
**映上函数/满射函数**是指对于函数，每个都有元素使得f(a) = b。



|  |
| --- |
| **范例：映上函数/满射函数** |
| , f(x) = x + 1是映上函数。  , f(x) = x2不是映上函数，因为没有整数x使x2 = -1。 |

**一一对应函数(One-to-One Correspondance)/双射函数(Bijection)**

如果一个函数**既是一对一函数又是映上函数**，那么这个函数就被称为**一一对应函数/双射函数**。



|  |
| --- |
| **范例：一一对应函数/双射函数** |
| f是从{a, b, c, d}到{1, 2, 3, 4}的函数，定义f(a) = 4，f(b) = 2，f(c) = 1，f(d) = 3。  函数f是单射函数，因为没有两个值映射到相同的函数值。  函数f是满射函数，因为陪域的个数与值域的个数相同。  因此，函数f是双射函数。 |

**3.4 反函数**

**反函数(Inverse Function)**

假设有一个从集合A到集合B的双射函数f。由于f是满射函数，所以B中的每个元素都是A中某些元素的像；又由于f还是单射函数，所以B的每个元素都是A中唯一一个元素的像。

于是，通过把f的**对应关系颠倒**，获得的从B到A的新函数被称为f的**反函数**，用“”表示。**当时，(b) = a**。

需要注意，不要将与混淆。

|  |  |
| --- | --- |
| **范例：反函数** | |
|  | |
| 该函数是否有反函数？ | 是 |
|  | a |
|  | d |

|  |
| --- |
| **范例：计算f(x) = x + 3的反函数** |
|  |

**3.5 合成函数**

**合成函数(Composition Function)**

假设g是从集合A到集合B的函数，f是从集合B到集合C的函数。函数f和g的**合成**，记作“”。

**函数合成的顺序很重要，与并不相等。**

|  |
| --- |
| **范例：合成函数** |
|  |

**恒等函数(Identity Function)**

如果一个从集合A到集合B的函数f有反函数，那么与的合成函数得到的是**恒等函数**。

|  |
| --- |
| 如果f(a) = b，那么。 |

**3.6 指数函数/对数函数**

**指数函数(Exponential Function)**

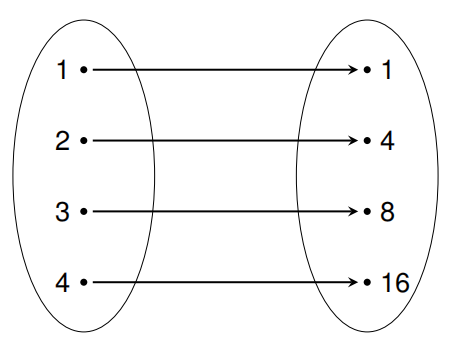
**指数函数**的定义为，其中a称为**底数**(base)，x称为**指数**(exponent)。

指数函数满足以下运算法则：

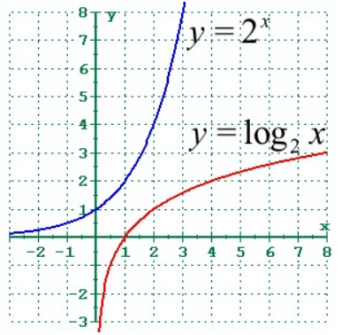
|  |
| --- |
| **范例：指数函数** |
|  |

**对数函数(Logarithm Function)**

对于函数，，指数函数是**双射函数**，因此它是有**反函数**的。



**对数函数是指数函数的反函数**，对数函数的定义为，其中a称为**底数**。



对数函数满足以下运算法则：

|  |
| --- |
| **范例：对数函数** |
|  |

**第4章 数论**

**4.1 进制转换**

**进制**

日常生活中都用**十进制(decimal)**来表示整数，十进制数由0, 1, 2, 3, 4, 5, 6, 7, 8, 9这10个字符组成。一个十进制整数的第k位的值可以由计算得到。

|  |
| --- |
| **范例：十进制** |
|  |

**二进制(binary)、八进制(octal)、十六进制(hexadecimal)**也是非常常用的表示法，例如**计算机通常用二进制来做算术运算**，而**用八进制或十六进制来表示字符**。

|  |  |  |  |
| --- | --- | --- | --- |
| **十进制** | **二进制** | **八进制** | **十六进制** |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 2 | 10 | 2 | 2 |
| 3 | 11 | 3 | 3 |
| 4 | 100 | 4 | 4 |
| 5 | 101 | 5 | 5 |
| 6 | 110 | 6 | 6 |
| 7 | 111 | 7 | 7 |
| 8 | 1000 | 10 | 8 |
| 9 | 1001 | 11 | 9 |
| 10 | 1010 | 12 | A |
| 11 | 1011 | 13 | B |
| 12 | 1100 | 14 | C |
| 13 | 1101 | 15 | D |
| 14 | 1110 | 16 | E |
| 15 | 1111 | 17 | F |

**进制转换**

一个b进制的正整数n可以唯一地构造**展开式**：

|  |
| --- |
| **范例：b进制展开式** |
|  |

|  |
| --- |
| **范例：b进制转十进制** |
|  |

十进制转b进制还可以使用**短除法**的方式。

|  |
| --- |
| **范例：十进制转b进制** |
|  |

**4.2 素数**

**素数(Prime Number)**

基于**整除性**的一个重要概念就是素数，**素数是大于1的且不能被1和它自身以外的正整数整除的整数**。素数是**现代密码学**中必不可少的一部分，密码学中的大素数就用在**信息加密**的某些方法中。

|  |
| --- |
| **范例：素数** |
| 7是素数，因子是1和7。  9是合数，9能被3整除。 |

每个大于1的整数都可以唯一地写成多个素数的乘积。

|  |
| --- |
| **范例：素因子分解式** |
| 100 = 2 \* 2 \* 5 \* 5 = 22 \* 52  999 = 3 \* 3 \* 3 \* 37 = 33 \* 37  1024 = 210 |

**如果n是一个合数，那么n必有一个素因子小于或等于。**

|  |
| --- |
| **范例：证明101是素数** |
| 不超过的素数只有2, 3, 5, 7，因为101不能被2, 3, 5, 7整除，所以101是素数。 |

|  |  |  |
| --- | --- | --- |
| **范例：判断素数** | | |
| import math  *def* is\_prime(*num*):      for i in range(2, *int*(math.sqrt(num)) + 1):          if num % i == 0:              return False      return True  *def* main():      print(is\_prime(13))      print(is\_prime(18))  if \_\_name\_\_ == "\_\_main\_\_":      main() | | |
| **运行结果** | True  False |

**埃拉托斯特尼筛法**(Sieve of Eratosthenes)可以用来寻找不超过一个给定整数的所有素数。步骤如下：

1. 建立包含所有给定整数以内的表格。
2. 从i = 2开始。
3. 移除所有整数n % i == 0（除i以外）。
4. i = i + 1。
5. 重复第3步和第4步。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 2 | 3 | ~~4~~ | 5 | ~~6~~ | 7 | ~~8~~ | ~~9~~ | ~~10~~ |
| 11 | ~~12~~ | 13 | ~~14~~ | ~~15~~ | ~~1~~ | 17 | ~~18~~ | 19 | ~~20~~ |
| ~~21~~ | ~~22~~ | 23 | ~~24~~ | ~~25~~ | ~~26~~ | ~~27~~ | ~~28~~ | 29 | ~~30~~ |

**4.3 序列**

**序列(Sequence)**

**序列**是一种用来表示**有序列表**的离散结构。例如1, 2, 3, 5, 8是一个含有五项的序列，而1, 3, 9, 27, 81, ..., 3n, ...是一个**无穷序列**。序列可以用记号“”表示。

|  |
| --- |
| **范例：序列** |
| 序列，其中：  序列，其中：1, 2, 4, 8, 16, 32, ... |

如果一个序列任意相邻的两项满足，那么这个序列被称为**递增序列**(increasing sequence)。

如果一个序列任意相邻的两项满足，那么这个序列被称为**非递减序列**(non-decreasing sequence)。

如果一个序列任意相邻的两项满足，那么这个序列被称为**递减序列**(decreasing sequence)。

如果一个序列任意相邻的两项满足，那么这个序列被称为**非递增序列**(non-increasing sequence)。

**算术级数(Arithmetic Sequence)**

**算术级数**也称“**等差级数**”，序列形式如下：

|  |
| --- |
| **范例：算术级数** |
| 序列，其中：-1, 3, 7, 11, ...  序列，其中：7, 4, 1, -2, ... |

**几何级数(Geometric Sequence)**

**几何级数也**称“**等比级数**”，序列形式如下：

|  |
| --- |
| **范例：几何级数** |
| 序列，其中：1, -1, 1, -1, 1, ...  序列，其中：2, 10, 50, 250, 1250, ...  序列，其中： |

**4.4 递推关系**

**递推关系(Recurrence Relation)**

如果数列的第n项与它前一项的关系可以用一个式子来表示，那么这个公式就叫做这个数列的**递推公式**。

算术级数的递推关系如下：

几何级数的递推关系如下：

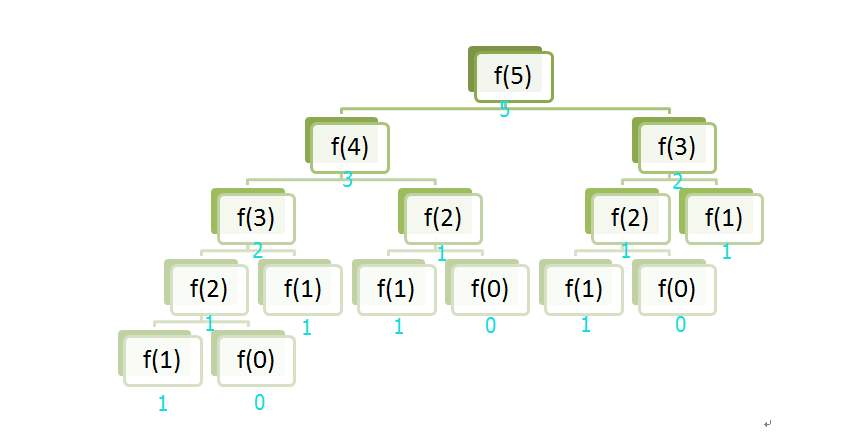
|  |
| --- |
| **范例：递推关系** |
| 是一个序列，满足递归关系 |

**斐波那契数列(Fibonacci Sequence)**

**斐波那契数列**，由初始条件递推关系定义。

|  |
| --- |
| **范例：斐波那契数列** |
|  |

|  |  |
| --- | --- |
| **范例：斐波那契数列（递归）** | |
| *def* fibonacci(*n*):      if n == 1 or n == 2:          return 1      return fibonacci(n-2) + fibonacci(n-1)  *def* main():      n = 7      print("斐波那契数列第%d位：%d" % (n, fibonacci(n)))  if \_\_name\_\_ == "\_\_main\_\_":      main() | |
| **运行结果** | 斐波那契数列第7位：13 |



|  |  |
| --- | --- |
| **范例：斐波那契数列（迭代）** | |
| *def* fibonacci(*n*):      f = [0] \* n      f[0] = f[1] = 1      for i in range(2, n):          f[i] = f[i-2] + f[i-1]      return f[n-1]  *def* main():      n = 7      print("斐波那契数列第%d位：%d" % (n, fibonacci(n)))  if \_\_name\_\_ == "\_\_main\_\_":      main() | |
| **运行结果** | 斐波那契数列第7位：13 |

|  |
| --- |
| **范例：银行储蓄账户上有10000元，年利率为5.8%，7年后账户中将有多少钱？** |
|  |

**4.5 求和**

**求和(Summation)**

求和记号“”可以用于表示序列中所有项的累加和。

在中，i用于表示求和**下标**，lower为**下限**，upper为**上限**。

|  |
| --- |
| **范例：求和** |
|  |

**双重求和**

很多情况下需要使用双重求和，比如在计算机程序中嵌套循环的分析中。

计算双重求和的方法是先展开内层求和，再继续计算外层求和。

|  |
| --- |
| **范例：双重求和** |
|  |

**4.6 数学归纳法**

**数学归纳法(Mathematical Induction)**

**数学归纳法**是一种数学证明方法，通常被用于证明某个给定命题在一个给定范围内成立。

数学归纳法分为三个步骤：**归纳基础**、**归纳假设**、**归纳递推**。

|  |
| --- |
| **范例：证明** |
| 1. 归纳基础：假设n = 1。  2. 归纳假设：假设n = k时，  成立。  3. 归纳递推：证明n = k+1时，  成立。 |

|  |
| --- |
| **范例：证明** |
| 1. 归纳基础：假设n = 4。  2. 归纳假设：假设n = k ()时，  成立。  3. 归纳递推：证明n = k+1时，  成立。 |