递归和循环

- 递归定义的函数，形式上是一个简单分支型程序
  - 每次递归调用执行函数里的一条执行路径
  - 从总体而言，可能多次重复执行函数里的语句
  - 运行中总的执行路径长度：由递归调用这个函数的次数确定，即是由调用函数的实参决定
- 理论研究结果
  - 由基本语句、顺序组合、条件语句、递归、函数定义构成的程序语言已经足够强大，足以描述所有可能的计算
  - 基本语句、顺序组合和循环构成的语言也足够强大
- 实际情况
  - 在实用的编程语言里，还需要引进另一些机制 (控制机制和数据机制)，以方便实际的程序开发

典型的递归实例：Fibonacci 数列的计算

- **Fibonacci (斐波那契) 数列是一个重要的自然数序列，在计算领域有重要的应用**
- **Fibonacci 数列的数学定义**

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2}, \qquad n > 1 \end{cases}$$

序列的前几项：**0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...**

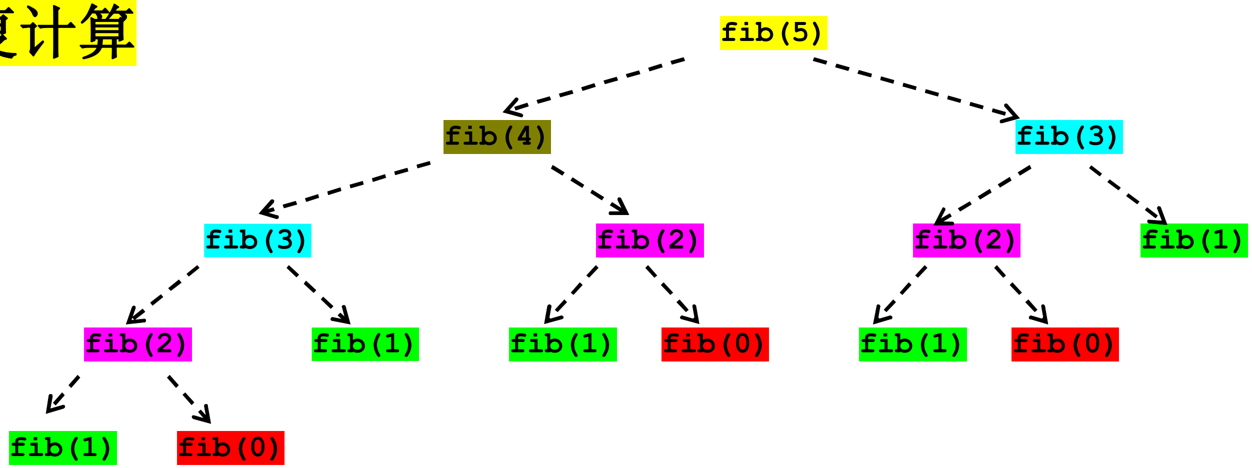
- 属于递归定义 (二路递归)

```python
def fib(n):
    if n < 1:              # 对负参数的处理
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

计算的时间

- 递归函数 fib 的定义直接对应于数学定义，简单清晰！
  - 但另一方面，存在着本质性缺陷：其计算要花费很长时间！
- 分析：计算 fib(5) 时，fib 被调用的情况

存在大量的重复计算



- 给计算过程计时：time 标准库包中的 time() 返回从 1970 年 1 月 1 日开始至今的秒数 (浮点数)

Fibonacci 数列的迭代/递推计算

- 分析 Fibonacci 数列的计算过程
  - 已知基本情况：F_0 和 F_1
  - F_{n+1}可以由 F_{n-1} 和 F_n 递推计算 (n > 1)

  典型的递推计算，从 F_0 和 F_1 出发逐个递推直至得到所需的 F_n
- 递推循环中涉及的变量
  - 用 f1、f2 记录已知的、两个相邻 Fib 项值
  - 用 f1 + f2 递推出下一个 Fib项值
  - 用变量 k 记录算到第几项，保证每次判断循环条件时 f1 的值总是第 k 个 Fib 项值
  - f1, f2 和 k 需要在每次循环中更新
  - 循环控制条件：k < n

```
f1, f2 =  f2, f1+f2
k += 1
```

用循环实现 Fibonacci 数列的递推计算

```python
def fib_loop1(n):
    if n <= 0: return 0

    f1, f2 = 0, 1  # 初始分别记录 F_0 和 F_1
    k = 0
    while k < n:
        f1, f2 = f2, f2 + f1  # 递推
        k += 1

    return f1
```

```python
def fib_loop2(n):
    f1, f2 = 0, 1          # 隐式地处理了 n < 0 的情况

    for k in range(0, n):
        f1, f2 = f2, f2 + f1

    return f1
```

循环不变式

- 循环结构中的循环体可能被很多次执行，如何保证计算的正确性？
  - 迭代过程中各个变量的值不断变化
  - 但是，某些变量之间的特定关系一直维持不变 — 循环不变式
  - 例如，考虑 **fib** 函数的递推实现 **(while** 语句**)**
    - 循环不变式：在每次循环判断时 **f1**、**f2** 的值正好是第 **k** 个和第 **k + 1** 个 **Fib** 项值
    - 循环结束 **(k = n)** 时， **f1** 即是第 **n** 个 **Fib** 项值
    - 这个结论具有一般性，与具体实参无关 **(**区别于用特定实参测试函数**)**，证明了函数的正确性
- 写好复杂循环的关键：确定在循环中应当维持哪些变量之间的何种关系不变，以保证当循环结束时关键变量均处于所需的状态

例：基于平方定义 power function (快速幂)

$$x^n = \begin{cases} 1 & \text{when } n = 0, \\ x \times x^{n-1} & \text{when } n \text{ is odd}, \\ (x \times x)^{n/2} & \text{when } n \neq 0 \text{ is even.} \end{cases}$$

```
def fastPower(x, n):
    if n == 0:
        return 1
    elif n%2 != 0: # 指数 n 是奇数
        return x * fastPower(x, n - 1)
    else:          # 指数 n 是偶数
        return fastPower(x * x, n // 2)
```

```
def fastPower_loop1(x, n):
    result = 1        # 累积变量，记录乘积结果
    while n > 0:      # 指数 n 为 0 时，循环终止
        if n%2 != 0:  # 指数 n 为奇数
            result *= x
            n -= 1
        else:         # 指数 n 为偶数
            x *= x    # x 自乘更新
            n //= 2
    return result
```

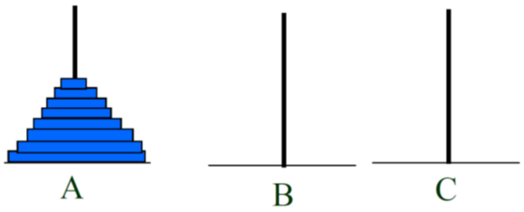
这里的循环不
变式是什么？

平方快速幂的改进实现

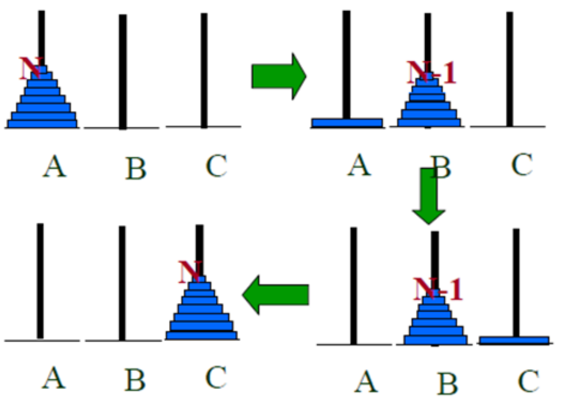
```python
def fastPower_loop2(x, n):
    result = 1
    while n > 0:
        if n%2 == 1:      # 指数 n 为奇数
            result *= x   # 更新 result
        x *= x            # x 自乘更新
        n //= 2
    return result
```

```python
def fastPower_loop3(x, n):
    result = 1
    while n > 0:
        if n & 1:     # 取指数 n 二进制表示的末位，并判断是否为 1
            result *= x
        x *= x
        n >>= 1       # 把 n (的二进制表示) 右移一位，相当于整除 2
    return result
```

递归实例：河内塔问题



- 有 3 根柱子 A、B、C，其中 A 柱上套有 64 个大小不等的圆盘，大盘在下，小盘在上。要求：
  - 要将 64 个圆盘从 A 柱搬到 C 柱
  - 每次只能搬动一个圆盘，搬动可以借助 B 柱进行
  - 在任何时候任何柱上的圆盘都必须保持大盘在下，小盘在上
  - 写程序模拟搬动圆盘的过程 (即输出搬动圆盘的步骤)
- 递归处理：搬 N 个圆盘可归结为搬 N-1 个圆盘
  - 借助 C 将 N-1 个圆盘从 A 搬到 B
  - 从 A 搬 (最下面的) 一个圆盘到 C
  - 借助 A 将 N-1 个圆盘从 B 搬到 C
- 实现细节：递归函数的参数？



递归实例：兑换硬币

- 人民币硬币有 1 分、2 分、5 分、10 分、50 分和 100 分。给定一定款额，问将其换成硬币有多少种不同的兑换方式
- 递归求解思路：款额 n 的总兑换方式数 =
  - 使用某一种硬币 a 之后，款额 n – a 的兑换方式数 +
  - 不用硬币 a 时，款额 n 的兑换方式数
- 有几种 (基础) 情况可以直接得到结果：
  1. n=0，说明找到一种兑换方式 (计 1 种兑换方式)
  2. n<0，说明没找到兑换方式 (计 0 种兑换方式)
  3. 硬币种类数=0，说明没找到兑换方式 (计 0 种兑换方式)
- 设计递归函数的参数：(1) 需要兑换的款额、(2) 可用的硬币 (币值)
  - 6 种币值无规律，无法直接递归！考虑定义辅助函数，编号不同币值的硬币 (即将编号 1 到 6 映射到 1 分到 100 分的硬币)，减少一种硬币时编号范围缩小一；也可利用表

用表记录递归过程的中间结果

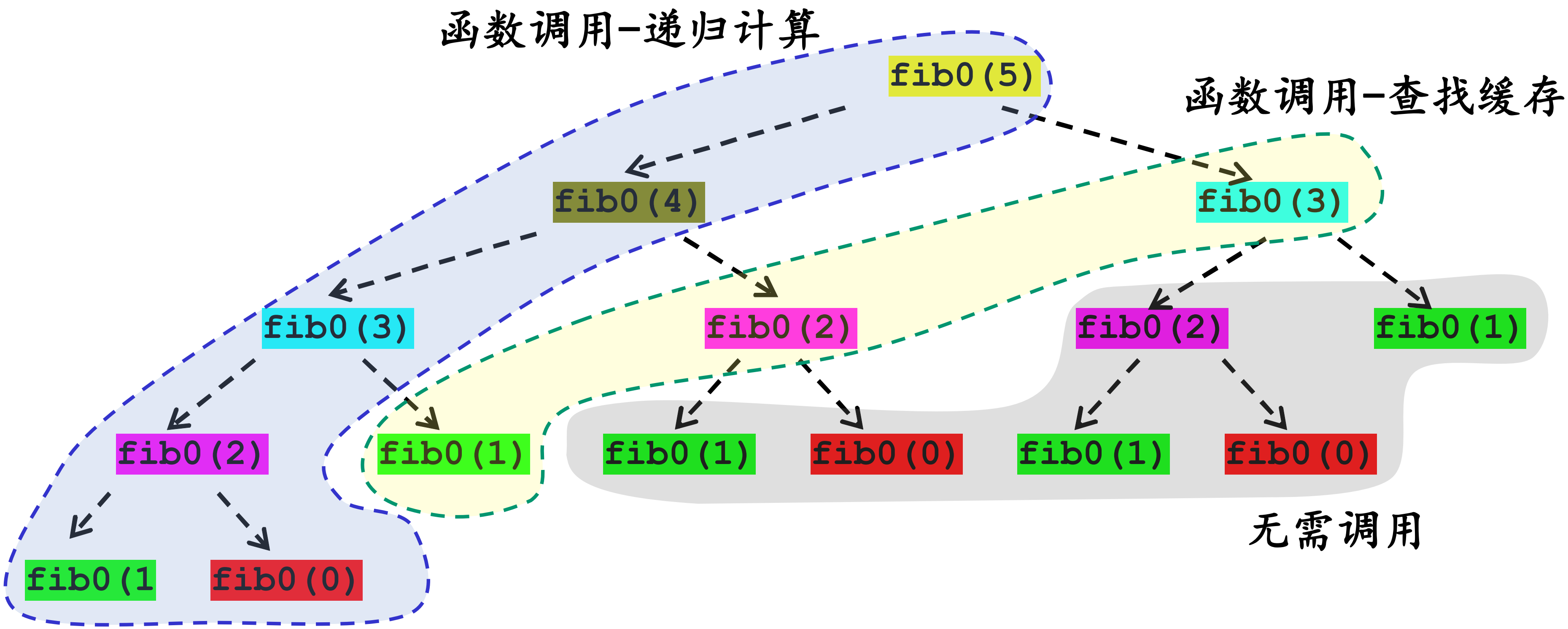
- 回顾：递归实现的 fib 函数的缺陷
  - 函数执行中存在大量重复计算 — 多次重复计算各 F_k 的值
  - 用一个缓存结构记录已经计算过的项值
  - 再次需要某个项值 F_k 时，通过 k 对缓存进行检查和取用
  - ➔ 效率的提高 (时间复杂性) vs 存储的代价 (空间复杂性)
- 实现细节：
  - 缓存的大小，依赖于实际调用 fib(n) 时 n 的实参值 (即在定义函数时并不确定)
  - 要通过 k 来检查、修改、或者取用缓存中的值
  - ➔ 适合用表作为缓存，用其元素记录已经计算出的 F_i
  - ➔ 需要计算新项值时，先检查缓存表：
    - 如果已经算过，则直接取用；如果没有算过，则先递归计算，并在返回结果前，要先记录到缓存表中的相应位置

记忆递归型 fib 函数 (1)

```
# 利用缓存表记录递归过程中的中间结果 ==> “以空间换时间”

def fib(n):
    fibs = [-1] * (n + 1)    # 缓存表 fibs (元素初值置 -1 或其他合理值)
    fibs[0], fibs[1] = 0, 1 # 设置表中前两项元素值为 Fib_0 和 Fib_1

    def fib0(k):             # 递归定义的局部函数，计算项值 Fib_k
        if fibs[k] != -1:    # 检查表 fibs (非局部变量) 中下标为 k 的项值
            return fibs[k]   # 如值非 -1，则直接取用 (即命中缓存)

        # 否则，递归计算出项值 Fib_k，并记录到到缓存表的相应项
        fibs[k] = fib0(k - 1) + fib0(k - 2)

        return fibs[k]

    return fib0(n)
```

局部函数的体可以使用其外围变量（详细情况之后介绍）

记忆递归型 fib 函数 (2)



计算 fib(5) 时的函数调用情况

复杂的递归情况

- 自递归：一个函数在体中调用自身
- 相互递归：两个 (或多个) 函数相互调用
  - 问题：如何排列相互递归调用函数的定义？

```python
def even(n):
    if n == 0: return True
    else: return odd(n - 1)

even(4)

def odd(n):
    if n == 0: return False
    else: return even(n - 1)
```

```
    else: return odd(n - 1)
NameError: name 'odd' is not defined
```

- **Python** 程序里，函数的定义可以任意排列
  - 在处理定义函数时，解释器不检查所用到函数有无定义
  - 在实际执行函数时，其中用到的所有函数必须都有定义

程序终止性

- 写程序时，需要考虑程序是否一定终止
  - 即，程序对所有可能的输入都能结束，函数对所有可能的参数都能完成计算并给出结果
  - 一方面，即使每条基本语句终止，循环 **(**递归**)** 程序也可能不终止
  - 另一方面，有些程序可能需要运行很长时间 **(**非死循环**/**非无穷递归**)**

  实例：计算调和级数 $\sum_{n=1}^{\infty}\frac{1}{n}$ 的前多少项之和能超过某个给定的正实数 **(17, 18, 19……** 详见演示**)**

- 有关程序终止性的理论结果：程序终止性问题是不可判定的 —— 图灵
  - 不存在判断任何程序对任何输入是否终止的有效方法
  - 直观说，就是无法写出一个程序，判断任意的一个程序对任意一个输入是否终止

程序终止性不可判定

- 例：**Collatz 猜想 (conjecture)**，猜想下面函数对所有正整数 **n** 终止 **(Ref: http://en.wikipedia.org/wiki/Collatz_conjecture)**

$$\text{collatz}(n) = \begin{cases} 1 & n = 1 \\ \text{collatz}(3n+1) & n \bmod 2 \neq 0 \\ \text{collatz}(n/2) & n \bmod 2 = 0 \end{cases}$$

- **Python 实现：**

```
def collatz(n):
    while n != 1:
        if n % 2 == 0 :            # n is even
            n =  n//2
        else:                      # n is odd
            n = n * 3 + 1
    return 1
```

- 可用不同的参数试验该函数的终止性，并可考察它迭代多少次，或达到的最大数值等；但是，无法严格证明

函数抽象的意义 (1)

- 作用 1：可能缩短程序
  - 如果函数的定义体比较长，而且在程序里多处调用
  - 定义函数只写一次，调用代码很短
- 作用 2：把一种有用的计算功能集中到一个地方描述，发现错误可以只在一个地方修正，也有利于今后的程序修改
  - 重要编程原则 (唯一定义原则)：程序中的任何重要功能都应该只有一个定义
- 作用 3：定义函数是在引入新的编程概念，扩充编程语言
  - 编程语言是通用的，只提供基本的概念和结构；
  - 利用函数机制，可以逐步建立一系列有用的新概念 (功能)，从而解决复杂问题 (自下向上的程序开发)

函数抽象的意义 (2)

- 作用 4：实现功能分解，分解程序的复杂性
  - 把要开发的复杂功能分解为一些概念清晰，功能较为简单的待开发函数 (自顶向下的程序开发，逐步求精)
  - 利于理清程序的实现结构，分解困难
- 作用 5：以函数为独立开发的单元
  - 方便开发的分工
  - 依赖于局部开发、检查、调试、发现和更正错误
- 作用 6：函数具有独立性和通用性
  - 经过严格检查和优化的通用功能有可能重用 (Python 的标准库和其他库是这方面的典型)
- (还有很多可能的作用)

程序的函数分解

- 基于函数的分解是 Python 程序最基本的功能分解，可实现最简单的模块化/结构化编程
- 哪些程序片断应该定义为函数？
  - 重复出现的相同或相似的代码片段
  - 具有逻辑独立性的片段
- 经验准则：如果一段计算可以定义为函数，则应该定义为函数！
- 函数定义的具体工作
  - 确定功能
  - 选定参数：将针对具体数据的计算抽象为相对一组参数的计算
  - 为函数命名
  - 选择适当的计算方法，实现函数体

看待函数的两种角度及其联系

黑盒子观点

函数的外部观点	头部内部与外部的联系	函数的内部观点
关心函数的使用 ● 实现了什么功能 ● 名字是什么 ● 要求几个参数，各参数的意义和作用 ● 返回什么值 ● 等等	头部 内部 与 外部 的 联系	关心函数的定义 ● 采用什么计算方法 ● 采用什么实现结构 ● 实际参数如何使用 ● 怎样得到所需要的返回值 ● 等等

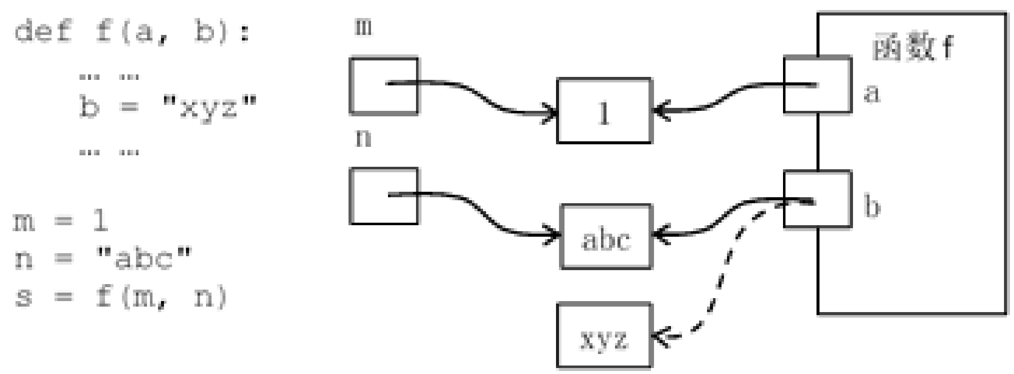
- 函数定义和使用应对函数功能有统一的理解，并遵循共同的规范
- 函数头部描述了函数内部和外部之间的联系，是交换信息的接口

函数的定义

- 定义函数需要设计函数头部
  - 函数头部是函数与外界的接口，是函数定义和调用共同遵循的规范
  - 细节：给函数命名；确定函数的参数 (各自的作用和类型)
- **Python** 函数定义的头部无法描述对形式参数的具体要求
  - 实际上，大多数函数对所需参数有特殊要求 (类型/值)
  - 例如：
    - 要求某参数是整数，或数值，或是函数等 (类型要求)
    - 要求某参数的值必须为正 (值要求)
  - 可用文档串说明具体要求，也可以用断言语句实施要求，或者用条件语句检查要求 (后面还将介绍其它处理机制)

函数的调用

- 调用函数时，必须提供合适的实参
  - 数量正确、类型和值均满足函数的要求
- 执行函数调用时
  - 解释器从左到右求值实参表达式，得到一组结果对象
  - 对应的函数形参以这些对象为值
  - 执行函数体
- 当实参表达式是另一个函数调用表达式，解释器将先完成那个函数调用，把返回的结果对象作为当前函数调用的实参



函数调用中形参和实参之间的关系

  - **Python** 允许写任意深度的函数 **(**嵌套**)** 调用

函数定义与调用的关系

- 全函数：对于任意类型合适的实参，函数总能计算得到正确的结果/完成所需的工作
- 对于不完全的函数，定义和调用之间必须相互配合
  - 定义函数时尽可能完全，用合理方式处理各种特殊情况
    - 断言语句可以明确描述程序运行 (函数执行) 必须满足的条件，有利于及时发现和报告错误，并方便定位错误
    - 带表达式的断言语句可以提供发生错误的详细现场信息，有助于确定错误原因
  - 在函数调用之前检查实参，保证符合函数要求再调用，对于不满足要求的数据另行处理 (可以用 if 语句)
  - 函数调用之后检查结果，确定返回值正确再使用，不正确的情况另行处理

函数的参数：带默认值形参

- 很多内置函数和库函数的部分形参有默认值
  - 调用时，可以不给带默认值的形参提供实参，解释器自动使用其默认值 (缺省值)
  - 注意：Python 手册描述函数形参时，用方括号括起来的部分在调用时可以缺省的

`input([prompt])`

If the *prompt* argument is present, it is written to standard output without a trailing newline.

`math.log(x[, base])`

With one argument, return the natural logarithm of *x* (to base *e*).

With two arguments, return the logarithm of *x* to the given *base*, calculated as `log(x)/log(base)`.

class `complex([real[, imag]])`

Return a complex number with the value *real* + *imag**1j

print 的默认参数

print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

- **单星号形参 *objects：表示调用时可以有任意多个任意类型的实参表达式，其值为实际输出的对象 (之后有更详细说明)**
- **带默认值形参 sep：表示两项输出之间的分隔串，默认为一个空格**
- **带默认值形参 end：表示输出完成后的结束串，默认为一个换行符**
- **带默认值形参 file：表示输出位置，默认输出到屏幕窗口**
- **调用 print 时，需采用关键字实参，即“形参名=表达式”的实参形式，为某个 (些) 带默认值形参提供非默认的值**

```
###### print 函数的使用示例
for i in range(11, 91):
    print(i, end = ", " if i%10 != 0 else "
```

```
11, 12, 13, 14, 15, 16, 17, 18, 19, 20
21, 22, 23, 24, 25, 26, 27, 28, 29, 30
31, 32, 33, 34, 35, 36, 37, 38, 39, 40
41, 42, 43, 44, 45, 46, 47, 48, 49, 50
51, 52, 53, 54, 55, 56, 57, 58, 59, 60
61, 62, 63, 64, 65, 66, 67, 68, 69, 70
71, 72, 73, 74, 75, 76, 77, 78, 79, 80
81, 82, 83, 84, 85, 86, 87, 88, 89, 90
>>> |
```

关键字实参和自定义带默认值形参的函数

- 调用函数时，关键字实参，能显式地指定实参表达式 (的值) 和形参之间的约束关系

```
>>> complex(imag=4, real=2)        # 等价于 complex(2, 4)
```

  - 所有关键字实参，必须写在其他普通实参 (即位置实参) 之后
  - 多个关键字实参可任意排列，但不能重复出现同一关键字

- 自定义带默认值形参的函数
  - 函数定义的形式参数列表中，可用“形参名=表达式”为指定形参提供默认值
    - 所有带默认值形参排在无默认值形参之后
  - 定义时，解释器求值默认值表达式，并记录形参名与表达式值的约束关系；调用时，如没为这种形参提供实参，则使用默认值

```
def f(n=0):
    print(n)
# try f(10), f(n=5), f()
```

Python 对模块测试的支持

- Python 程序运行中，总有一个全局变量 `__name__`
  - 在程序执行的每个时刻， `__name__` 的值都是一个字符串
  - 当通过导入的方式执行一个模块 (py 文件)，在该模块执行期间， `__name__` 的值为该 py 文件的名字 (一个字符串)
  - 当一个 py 文件作为主模块启动时， `__name__` 值为特殊字符串 `"__main__"`
- 编程习惯：在一个模块 (py 文件) 里，可以写一段只在本模块用作主模块时才执行的代码 (即测试代码)

```
if __name__ == "__main__" :
    # 作为主模块时才执行的代码
    …
    …
```

实例：求立方根近似值 (通用方法 1)

- 问题：定义函数计算任一浮点数的立方根 (近似值)，比如要求所得根的立方与原数的差的绝对值不超过 0.001 (绝对误差)
- 通用方法 1：生成 (枚举) 一系列数值做试验，晒选出最接近的值
  - 如果试验的数值足够密集，就可能得到足够好的解
  - 可以按照不同的步长 (如 0.001 等) 做试验
- 演示代码和一些细节
  - 将负数求根归结到正数，统一处理
  - 搭建测试平台进行黑箱测试：交互式、自动 (或随机) 测试等
- 测验结果：用固定步长检查，未必能保证对所有可能数值找到的根都满足要求 (对绝对值较大的浮点数都找不到)
  - 反思计算方法：步长与结果，步长 (精度) 与计算时间

实例：求立方根近似值 (通用方法 2)

- 通用方法 2：逼近方法
  - 选取一个包含解 (立方根) 的区间
  - 逐步缩小区间范围，而且保证解包含在其中
  - 当区间足够小时，即可用其中点作为解的近似值
- 细节：
  - 初始区间的选取？(要求：包含解、值单调)
  - 如何缩小区间？
    - 任何能保证不丢掉解的方法都可以考虑
    - 二分 (折半) 法：每次二分区间后转到包含解的半区间，反复二分区间可以变得任意短，从而得到任意精度的解

演示程序

实例：求立方根近似值 (专用方法)

- 通用方法具有广泛的适用性，但解决问题的效率相对较低
- 利用牛顿迭代法：立方根逼近公式

$$x_{n+1} = \frac{1}{3}(2x_n + \frac{x}{x_n^2})$$

目标：达到精度

$$\left|(\frac{x_{n+1} - x_n}{x_n})\right| < 10^{-6}$$

- 细节
  - 从参数 x 开始迭代
  - 需要前后两个迭代值，以便判断
  - 收敛性由理论保证