**Модуль 3. Функции, модули, файлы. Раздел 3-1 (7 часов)**

*Тема 1.1. Работа с функциями.* ***Лекционное занятие (1 час)***

Функция – именованный фрагмент программного кода, к которому можно обратиться из другого места программы (есть и «безымянные» **lambda**-*функции*). Функции служат для работы с данными, передаваемыми как аргументы. Из каждой функции можно сформировать возвращаемое значение, но его может и не быть. Для удобства функции, не возвращающие значения, называют ПРОЦЕДУРАМИ, а всё множество процедур и функций называют ПОДПРОГРАММАМИ.

Функция - вспомогательный алгоритм для многократного обращения в программе к одному и тому же фрагменту, а также служащие для разделения программы на компоненты – то есть выполнения *процедурной декомпозиции*.

В общем виде функция имеет следующий формат:

**def** <имя\_функции>(par1, par2,... parN):

 операторы

Описание функции начинается с заголовка, где инструкция **def** определяет ее имя, далее, в круглых скобках - список параметров, который может быть и пустым. Параметры в заголовке связаны с аргументами функции порядком следования.

Стоит отметить, что аргументы часто путают с параметрами:

* параметр – переменная, которой *будет* присваиваться входящее в функцию при ее вызове значение, а аргумент – САМО ЭТО ЗНАЧЕНИЕ, указанное при вызове.

Часто оба списка, и указываемый при вызове функции и в ее заголовке, называют ПАРАМЕТРЫ, добавляя уточнение: «*фактический*» и «*формальный*».

Возврат значения осуществляют оператором **return**, указав после него возвращаемый объект: **return** <выражение> (уголковые скобки нигде не ставятся).

Оператор **return** располаают в любом месте функции, он завершает ее работу и передает результат вызывающей программе. Если **return** отсутствует, то работа функции завершается достижением конца ее тела. Пример вызова функции:

<имя\_функции>(arg1, arg2,... argN), или же в составе выражения.

Количество аргументов и параметров при вызове функции обычно совпадает, если не запрограммировать переменное количество принимаемых аргументов. В качестве аргументов могут выступать как конкретные значения, так и переменные.

Таким образом, общая структура работы с функцией имеет вид (Рисунок 1).

|  |
| --- |
| **def** <имя\_функции>(par1, par2,…,parN) |
| Тело |
| функции |
|  |
| <имя\_функции>(arg1, arg2,…,argN) |

Рис. 1 – Схема работы с функцией

Приведем два примера описания и вызова простых функций:

**def** s1(X,Y):

 **return** X\*\*2+Y

X=**float**(**input**()); Y=**float**(**input**()); **print**(s1(X,Y))

Здесь описана функция вычисления $x^{2}+y$. В главной программе с клавиатуры вводят 2 вещественных числа: (X, Y), а вывод значения функции s1, где аргументы числа(X, Y) – на экран. Результат работы программы приведен ниже (Рис. 2):

|  |  |
| --- | --- |
|  | Рис.2 – Результат работы программы |
| **def** mmm(a,b,c): **if** (a>b) **and** (a>c): **print**(a) **elif** (b>c) **and** (b>a): **print**(b) **else**: print(c)mmm(3,6,1) | Функция mmm не возвращает результат (то есть это процедура), но операторы **print** выводят результат в ее теле. Обращение к процедуре происходит не в составе выражения или другого оператора, а отдельной строкой. |

Как уже отмечалось, для функций программист может использовать различное количество аргументов. Функция также может принимать переменное количество *позиционных* аргументов, тогда перед именем ставится \*:

**def** func(\*pars): #вместо pars («параметры») можно указать любое имя

 **return** pars

Если в этом случае требуются параметры с именами, тогда перед именем ставят \*\*:

**def** func(\*\*kwpars): #kwpars-«параметры с ключевыми словами» - имя также любое

 **return** kwpars

В этом случае аргументы передаются как словари. Более подробное описание будет дано в конце раздела «Функции».

Также стоит отметить вопрос о видимости переменных. Область видимости в Python-е представляет собой пространство имен, где функционируют созданные объекты. Это позволяет разграничивать доступ и избежать конфликтов имен. Есть локальные и глобальные переменные, что зависит от места их инициализации.

В Python *переменная, объявленная вне функции или в глобальной области видимости, называется глобальной переменной*. К глобальной переменной можно получить доступ как внутри, так и вне функции. Рассмотрим пример.

x = 5 #глобальная

**def** f1():

 **print**("x внутри функции:", x)

f1(); **print**("x вне функции:", x)

Здесь переменная x - глобальная. Результат работы функции – см. рис. 3:

|  |  |
| --- | --- |
|  | Рис.3 – Результат работы программы |

Переменная, объявленная внутри тела функции или в локальной области видимости, называется локальной переменной. Приведем следующий пример.

**def** f2():

 Y = 5; **print**(Y+1) #Y – локальная переменная

f2()

Здесь переменная Y – локальная, так как ее инициализация выполнена внутри f2.

Изменим пример.

**def** f2():

 Y = 5; **print**(Y+1)

f2(); **print**(Y)

Результат работы данной программы выглядит следующим образом (рис. 4).

|  |  |
| --- | --- |
|  | Рис.4 – Результат работы программы |

Как мы видим, Python выдает ошибку, потому что пытаемся получить доступ к локальной переменной Y в глобальной области видимости. Так делать нельзя: локальная переменная Y определена только внутри функции f2.

*Тема 1.1. Работа с функциями.* ***Практическое занятие 1 (1 час)***

1. Написать функцию, которая возвращает процент от числа.

**def** percent(x,n): # x – число, n-нужный процент от него

 **return** x/100\*n

x=**int**(**input**('Число:')); n=**int**(**input**('Процент:')); **print**(percent(x,n))

1. Написать функцию, которая находит наибольшее из целых чисел.

В задании не указано количество чисел, но можно *предположить* 2 числа, или же работать с переменным числом параметров (функцией **max** пользоваться не будем).

|  |  |
| --- | --- |
| Два параметра | Любое число параметров |
| **def** mm(x,y): **if** x>y: **return** x **else**: **return** y**print**(mm(5,9)) | **def** mm(\*x): mx=x[0] **for** t **in** x: **if** mx<t: mx=t **return** mx**print**(mm(3,4,5,2)) | **def** mm(\*x): **return** **max**(x)**print**(mm(3,5,9,7,4))#здесь сразу используем стандартную #функцию max, то есть для решения#задачи создавать функцию не нужно |

1. Написать функцию поиска НОД двух чисел.

Для функции можно использовать «первый» (разностный) или «второй» («быстрый») алгоритм Евклида, который он описал в III веке в своих «Началах». В модуле **math** есть функция **gcd**, которой пользоваться не будем.

|  |  |
| --- | --- |
| Первый алгоритм | Второй алгоритм – 2 варианта: |
| **def** nod(a,b): **while** a!=b: **if** a>b: a-=b **else**: b-=a **return** a | циклический | рекурсивный |
| **def** nod(a,b): **while** b!=0: a,b=b, a%b **return** a | **def** nod(a,b): **if** b!=0: **return** nod(b,a%b) **return** a |

Головная: x, y=**map**(**int, input**('НОД(a,b). Введи a,b:').**split**()); **print**(nod(x,y))

 Функция **gcd** библиотеки **math** ищет НОД с любым числом аргументов.

1. Напишите функцию для вычисления факториала числа.

Задачу также можно решить функцией **factorial** модуля **math**, но мы сделаем это самостоятельно. Как известно, факториал числа n вычисляется по формуле: $n!=1∙2…∙(n-1)∙n$. Хорошо видно цепное произведение чисел от 1 до n. Видно также, что $n!=1∙2…∙\left(n-1\right)∙n=n∙\left(n-1\right)!, то есть f\left(n\right)=n∙f(n-1)$. Здесь идет последовательное упрощение задачи уменьшением аргумента, получив, в конечном итоге: $f(1)=1$. Имеем два простых варианта вычисления функции:

|  |  |
| --- | --- |
| **def** f(n): #простой цикл p=n **for** i **in** **range**(2,n): p\*=i **return** p | **def** f(n): #рекурсивный алгоритм **if** n==1: **return** 1 **return** n\*f(n-1) |

Головная программа имеет вид: n=**int**(**input**('Вычисляем n! Введи n:')); **print**(f(n))

*Тема 1.2. Работа с функциями.* ***Лекционное занятие (1 час)***

Рассмотрим подробнее вопрос возврата функцией нескольких значений.

*Использование списка (list)*

Данная структура данных была описана ранее, в **Модуле2**. Список может быть использован в качестве результат работы функции. Приведем два примера:

|  |  |
| --- | --- |
| **def** f\_list(): x=15; y=20; z=-10; **return** [x,y,z]; **print**(f\_list()) | **def** f\_list(n): s=[] **for** i **in** **range**(1, 5): s+=[n\*\*i] **return** s **print**(f\_list(3)) |

Слева - результ работы функции f\_list – список из трех элементов (см. рис.1):

|  |  |
| --- | --- |
|  | Рис.1. Результат работы программы |

Результат справа – список, формируемый внутри функции f\_list (см. рис.2).

|  |  |
| --- | --- |
| Рис.2. Результат работы программы |  |

*Использование кортежа (tuple)*

Кортеж может быть результатом работы функции. Приведем пример.

**def** f\_tup():

 s = "s1"; x = 20; **return** s, x

s, x = f\_tup(); **print**(s, ‘\n’, x)

В данном примере результат работы функции – кортеж, который явно не обозначен.

|  |  |
| --- | --- |
|  | Рис.3. Результат работы программы |

*Использование словаря (dictionary)*

Словари также используют как результат работы функции. Рассмотрим пример.

**def** p\_age():

 d = **dict**(); d['Иван'] = 30; d['Петр'] = 28; d['Татьяна'] = 27

 **return** d

d = p\_age(); **print**(d)

Результатом функции является кортеж d, который содержит 3 значения.



Рис.4. Результат работы программы

В заключении отметим основные преимущества использования функций.

**Абстракция.** Можно уйти от конкретных значений к абстрактным объектам. Гораздо проще написать функцию, которая находит решение для множества однотипных, в рамках абстракции, объектов. В случае громоздких это повлечёт за собой значительное сокращение объёмов кода, а значит и времени на его написание.

**Возможность повторного использования кода.** Функции созданы ради их многократного применения. Без них код программы существенно больше, чем с функциями. Также облегчается возможность отладки и модификации программы.

**Модульность.** Деление больших задач на подзадачи – важная часть, как в кодинге, так и реальной жизни: когда уборка в квартире, то сначала пылесосим, затем моем полы и окна, очищаем поверхности от пыли и т.д.

Всё это – составляющие одного большого процесса под названием "уборка", но каждую из них также можно разбить на более простые подпроцессы.

*Тема 1.2. Работа с функциями.* ***Практическое занятие 2 (2 часа)***

Необходимо написать функции для решения следующих задач:

1. проверки введенного числа на «простоту».

Пусть результат работы функции – **True** или **False**. Покажем 3 типовых алгоритма:

|  |  |  |
| --- | --- | --- |
| «Медленный» $O(n)$ | «Быстрый» $O(\sqrt{n})$ | Рекурсия $O(\sqrt{n})$ |
| **def** prime(n): f= **True** **for** i **in** **range**(2, n//2+1): **if** n%i==0: f=**False**; **break** **return** f | **def** prime(n): i=2;f=**True** **while** (i\*i<n) **and** f: **if** n%i==0: f=**False** i+=1 **return** f | **def** prime(n, i)**:** **if** i\*i>n: **return True** **if** n%i==0**: return False** **return** prime(n, i+1) |

Вызов для столбцов 1-2: **print**(prime(997)), для столбца 3: **print**(prime(997, 2)).

|  |  |
| --- | --- |
| 1. по координатам концов отрезка поиск его длины
 | 1. поиска количества цифр целого числа X (считает только цифры!)
 |
| **def** L\_otr(x1,y1,x2,y2):  **return** ((x1-x2)\*\*2+(y1-y2)\*\*2)\*\*0.5**print**(L\_otr(3,5,1,1)) | **def** count(x): k=0 **for** c **in** x: **if** '9'>=c>='0': k+=1 **return** kx=**input(**'X='**); print**(count(x)) |

1. вывода гласных символов введенной строки.

**def** glas(a):

 **for** k **in** a:

 **if** k **in** 'ёуеыаоэяиюЁУЕЫАОЭЯИЮaeiouAEIOU': **print**(k)

a=**input**('Выводим гласные в данной строке. Введи строку:'); glas(a)

*Тема 1.3. Работа с функциями.* ***Практическое занятие 3 (1 час)***

1. Написать функцию сравнивающую 2 числа с результатом в виде знаков >, <, =.

**def** znak(a,b):

 **if** a>b: **return** '>'

 **elif** a==b: **return** '='

 **else**: **return** '<'

**print**(znak(4,9))

1. Написать функцию, которая возвращает n-ое число Фибоначчи.

|  |  |
| --- | --- |
| Рекурсивный алгоритм | Циклический алгоритм |
| **def** fib(n): **if** n<2: **return** 1 **else**: **return** fib(n-1)+fib(n-2) | **def** fib(n): a=1;b=1 **for** i **in** **range**(1, n): a,b=b, a+b **return** b |

Головная программа имеет вид: **print**(fib(4))

|  |  |
| --- | --- |
| 1. умножение чисел списка
 | 1. сумма цифр строки
 |
| **def** mul(s):  t = 1 **for** x **in** s: t \*= x **return** t**print**(mul((8, 2, 3, -1, 7))) | **def** sum\_c(s): sc=0 **for** x **in** s: **if** '9'>=x>='0': sc+=**int**(x) **return** sc**print**(sum\_c('12hjhk78')) |

*Тема 1.4. Работа с функциями.* ***Практическое занятие 4 (2 часа)***

1. Напишите функцию для обращения строки.

**def** revers(st):

 rst = ''

 **for** i **in** st: rst = i+rst

 **return** rst

**print**(revers('1234abcd'))

1. Напишите функцию подсчета в строке числа букв верхнего и нижнего регистра.

|  |  |
| --- | --- |
| Встроенными функциями | Через сравнение кодов символов |
| **def** str\_count(s): sl=0; sh=0 **for** c **in** s: **if** c.**islower**(): sl+=1 **elif** c.**isupper**(): sh+=1 **return** (sl,sh) | **def** str\_count(s): sl=0; sh=0 **for** c **in** s: **if** ('z'>=c>='a') **or** (c=='ё') **or** ('я'>=c>='а'): sl+=1 **elif** ('Z'>=c>='A') **or** (c=='Ё') **or** ('Я'>=c>='А'): sh+=1 **return** (sl,sh) |

s='Vfvf sdfasd KuyёhKAЁSD KJHамаKHHмылаGhgkМилуlhjgg ghKhg 14123ghkjjh'

d=str\_count(s); **print**(f'Lowercase:{d[0]}, Uppercase:{d[1]}')

Функция возвращает кортеж из количеств букв нижнего и верхнего регистра.

1. Напишите функцию, которая в заданном списке удаляет повторы.

|  |  |
| --- | --- |
| Не подсчитывает повторы + медленый | Метод считает повторы + быстрый |
| 123456 | **def** unicum(S): X = [] **for** a **in** S: **if** a **not in** X: X.**append**(a) **return** X**print**(unicum([5,2,3,7,3,3,4,])) | **def** unicum(S): X = **dict**.**fromkeys**(S,1) **for** a **in** S: X[a]+=1 **return** X**print**(unicum([5,2,3,7,3,3,4,5]).**keys**()) |

Слева – функция, работающая за **O(m\*n)** (**m, n** – мощность алфавита списка и его размер), что видно по строке 4, и без подсчета количества повторов каждого уникального значения списка. Справа – «продвинутая» функция, работающая за **O(n)** возвращая словарь, где массив ключей – заданный список без повторов (то, что нужно получить), а массив значений – число повторов каждого ключа.

1. Написать функцию, возвращающую четные значения заданного списка.

**def** lst\_even(s):

 snew=[]

 **for** x **in** s:

 **if** x%2==0: snew+=[x]

 **return** snew

**print**(lst\_even([1,6,7,9,10]))

*Тема 1.5 (дополнительная). Переменное число параметров: примеры.*

**Использование \* и \*\* в заголовке функции**

|  |  |
| --- | --- |
| **def** f1(\*x): **print**(x)f1(1,2,3,'4','56')На выходе-кортеж: (1, 2, 3, '4', '56') | **def** f1(\*x): **print**(x)f1([1,2,3,'4','56'])На выходе-кортеж: ([1, 2, 3, '4', '56'], ) |
| **def** prnt(\*\*x): **for** key, value **in** x.**items**(): **print**("{}:{}".**format**(key,value), **end**=’’)prnt(a="hot", b="cold",c="hard",d="soft")На выходе-словарь x по компонентам:  a:hot b:cold c:hard d:soft | t={} #глобальная переменная**def** prnt(\*\*x):  **print**(x); x['b']+=1; t.**update**(x)prnt(a="hot", b=2, c=True); **print**(t)На выходе-словари x и t:{'a': 'hot', 'b': 2, 'c': True}{'a': 'hot', 'b': 3, 'c': True} |

**Использование \* и \*\* при вызове функции**

|  |  |
| --- | --- |
| **def** f2(a, b, c): **print**("a:",a,";b:",b,";c:",c,**sep**='')x = ("cold", "hot", "warm"); f2(\*x)переменные из кортежа: a,b,c: a:cold; b:hot; c:warm | **def** f2(a,b,c,d): **print**("a:",a,";b:",b,";c:", c,";d:",d,**sep**='')x={"b":"cold","c":"hot","d":"warm"}; f2(1,\*\*x)Из числа и словаря – в 4 переменные:a:1; b:cold; c:hot; d:warm(при несовпадении имен – ошибка) |